UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

AUTOMATIC GENERATION OF SMELL-FREE UNIT TESTS

João Gonçalo Balsinha Afonso

Mestrado em Engenharia Informática

Dissertação orientada por:
Prof. Doutor José Carlos Medeiros de Campos

2022

Acknowledgments

First and foremost, I would like to thank my supervisor, Prof. José Carlos Medeiros de Campos, for his continuous support, availability, patience, commitment, and guidance. His feedback was essential to the success of this work.

I will be forever grateful to my family, in particular to my parents and brother, for their love and support. They were always there for me and gave me the strength I needed to continue pushing through. I would never have made it without them.

I would also like to thank all my friends who have supported me throughout these years. Special thanks to: João Martins, who has been my best friend since I was born; João Grilo, Vânia Romão, and Tiago Sobral for being the best friends I made during my school years and for being some of the kindest people I have ever met; Tiago Carvalho, Miguel Dias, Miguel Saldanha, and Alexandre Monteiro, whom I met during my years at FCUL and who have been not only the most amazing colleagues I could have but also the most amazing friends. I feel very fortunate that I have met all of you.

This work was supported by the Fundação de Ciências e Tecnologias (FCT) through the LASIGE research unit ((UIDB/00408/2020) – Ref.ª 716).

Dedicated to my family and friends.

Resumo

Testar corresponde a um processo indispensável para a geração de software de alta qualidade. Contudo, o desenvolvimento de testes de software é dispendioso. Portanto, com o intuito de mitigar estes custos elevados, têm sido desenvolvidas ferramentas que permitem gerar testes automaticamente (como é o caso de ferramentas como o EvoSuite e o Randoop). Estas ferramentas de geração automática de testes tendem a ser bastante eficazes a produzir testes com elevada cobertura (ou seja, testes que exercitam uma grande quantidade de código), mas frequentemente negligenciam a qualidade dos testes gerados. Como tal, testes gerados automaticamente estão frequentemente sujeitos a um conjunto de más práticas de programação que podem ter efeitos adversos não só na qualidade dos próprios testes, mas também na qualidade do código de produção: *test smells*.

Ferramentas automáticas como o EvoSuite não têm a capacidade de "adivinhar" qual é o comportamento suposto de um determinado programa. Nomeadamente, a razão pela qual se usa este tipo de ferramentas é para gerar testes que manifestam o comportamento atual do código de produção (seja ele, ou não, o correto). Assim sendo, os developers têm de analisar e perceber o código dos testes gerados para poderem averiguar se o comportamento atual de um dado programa está de acordo com o esperado. Por sua vez, se os testes gerados automaticamente utilizarem más práticas de programação, então a respetiva análise e manutenção será mais difícil. Em última instância, caso o tempo necessário para fazer a análise dos testes exceda o tempo que se poupou por fazer a geração automática dos mesmos (em vez de os escrever manualmente), então a utilidade destas ferramentas torna-se questionável (nesse caso, não faria sentido usar testes gerados automaticamente).

Tendo em consideração os problemas associados com a presença de smells em testes gerados automaticamente, comprometemo-nos a integrar um conjunto de métricas smell-free na ferramenta EvoSuite e a desenvolver uma abordagem para otimizar essas métricas de modo a minimizar a smelliness dos testes produzidos. Concomitantemente, assumimos que seria necessário alcançar este objetivo sem comprometer a cobertura ou a capacidade de detetar faults da ferramenta (caso contrário, na prática, a abordagem não seria útil).

Para este efeito, compilámos test smells de um conjunto tão vasto quanto possível de diferentes fontes e identificámos os smells relevantes para o contexto deste trabalho, ou seja, os smells que podem afetar os testes gerados pelo EvoSuite e que também podem ser

caracterizados por métricas otimizáveis. Posteriormente, definimos métricas representativas para os test smells escolhidos e integrámos esse conjunto de métricas na ferramenta EvoSuite. Por fim, foi feita uma análise de três abordagens distintas que, em teoria, deveriam possibilitar a otimização das métricas de test smells na ferramenta EvoSuite. Nomeadamente, ponderámos otimizar as métricas de test smells como: (1) um critério adicional; (2) critérios secundários; (3) passos adicionais de pós-processamento. Tendo sido feito um estudo destas três abordagens, tornou-se claro que a abordagem com maior potencial era aquela que consistia em otimizar métricas como critérios secundários.

Concretamente, optámos por substituir o critério secundário de default do EvoSuite (tamanho dos testes) por um conjunto de critérios secundários que permitem otimizar as métricas de test smells. De um modo geral, caso o EvoSuite esteja a comparar testes que são equivalentes do ponto de vista de cobertura, então são utilizados critérios secundários que dão prioridade aos testes que têm outras características apelativas. No nosso caso, isso implica que os testes gerados só são comparados em termos de métricas de test smells se forem equivalentes do ponto de vista da cobertura, ou seja, podemos otimizar as métricas de test smells sem interferir diretamente com a cobertura dos testes gerados.

A otimização de métricas de test smells como passos adicionais de pós-processamento também despertou o nosso interesse. O pós-processamento corresponde a um conjunto de otimizações feitas após a geração dos testes que visam melhorar diversos aspetos de qualidade dos mesmos (por exemplo, remoção de linhas de código desnecessárias). Contudo, tivemos de excluir esta opção porque não tínhamos acesso a uma funcionalidade que permitisse criar testes equivalentes, mas com uma estrutura fundamentalmente diferente.

Investigámos a difusão de test smells e a distribuição das respetivas métricas nos testes gerados pela versão default da ferramenta EvoSuite e observámos que os passos de pós-processamento reduzem imensamente a smelliness dos testes gerados. De qualquer modo, os testes gerados automaticamente continuam sujeitos a diversos tipos de más práticas de programação. Nomeadamente, os seguintes smells tendem a ser os mais difusos nos testes gerados pelo EvoSuite: “Unknown Test”, “Indirect Testing”, e “Unused Inputs”. Dados os smells que afetam os testes gerados pelo EvoSuite na prática, também identificámos aqueles que poderiam ser otimizados como critérios secundários.

Após identificarmos os tipos de test smells que, na prática, afetam os testes gerados pelo EvoSuite, fizemos um estudo de tuning para identificar a combinação ideal de test smells para otimizar como critério secundários, ou seja, a combinação que minimiza a smelliness dos testes tanto quanto possível e que maximiza a cobertura e a capacidade de deteção de faults tanto quanto possível. Em concreto, realizámos pairwise tournaments para as seguintes métricas: cobertura, mutation score, e smelliness. Dados os resultados dos torneios, fizemos o ranking das configurações de acordo as das três métricas referidas e, desta forma, acabámos por optar por usar a combinação que otimiza as seguintes métricas: “Eager Test”, “Indirect Testing”, “Obscure In-line Setup”, e “Verbose Test”.

Comparámos os testes gerados pela nova versão do EvoSuite com aqueles gerados pela versão original da ferramenta e observámos que, de facto, a smelliness dos testes gerados tinha melhorado. Nomeadamente, o smell "Indirect Testing" (que corresponde ao segundo smell mais difuso nos testes gerados pela versão original do EvoSuite) revelou melhorias significativas. Por sua vez, o smell "Unrelated Assertions", que está associado com o "Indirect Testing", também se tornou menos difuso. Tendo em consideração que substituímos o critério secundário de default do EvoSuite (que é equivalente ao "Verbose Test") por um conjunto de métricas de test smells, observou-se um aumento ligeiro no tamanho dos test cases. Embora a combinação de métricas de test smells que optámos por otimizar como critérios secundários também inclua o "Verbose Test", já não é dado um foco exclusivo apenas ao tamanho dos testes. Como tal, é normal que os test cases tendam a ser um pouco maiores na nova versão do EvoSuite.

Observámos que a otimização das métricas de test smells não comprometeu nem a cobertura nem a capacidade de detetar faults dos testes. De qualquer modo, é importante referir que, apesar de não ter sido uma redução significativa, a nova versão do EvoSuite é ligeiramente pior do que a original em termos de capacidade de detetar faults. Este resultado foi inesperado porque, de acordo com o estudo de tuning, a "combinação ideal de test smells" era a décima melhor configuração em termos de mutation score. Por sua vez, nenhuma das outras combinações de métricas que estavam entre as dez melhores configurações em termos de mutation score eram viáveis do ponto de vista de smelliness, ou seja, parece evidente que a otimização de métricas de test smells como critérios secundários pode ter ligeiros efeitos adversos no mutation score. Também constatámos que não houve qualquer variação significativa no número total de linhas de código ou no número de test cases nos test suites gerados pela nova versão do EvoSuite.

De um modo geral, podemos confirmar que fomos capazes de alcançar os nossos objetivos, ou seja, conseguimos tornar os testes gerados pelo EvoSuite menos smelly sem comprometer a cobertura ou a capacidade de detetar faults dos testes gerados.

Palavras-chave: Test Smells, Qualidade de Software, Geração Automática de Testes, Otimização de Múltiplos Objetivos, Estudos Empíricos

Abstract

Automated test generation tools (such as EvoSuite) typically aim to maximize code coverage. However, they frequently disregard non-coverage aspects that can be relevant for testers, such as the quality of the generated tests. Therefore, automatically generated tests are often affected by a set of test-specific *bad* programming practices that may hinder the quality of both test and production code, i.e., *test smells*. Given that other researchers have successfully integrated non-coverage quality metrics into EvoSuite, we decided to extend the EvoSuite tool such that the generated test code is smell-free. To this aim, we compiled 54 test smells from several sources and selected 16 smells that are relevant to the context of this work. We then augmented the tool with the respective test smell metrics and investigated the diffusion of the selected smells and the distribution of the metrics. Finally, we implemented an approach to optimize the test smell metrics as secondary criteria. After establishing the optimal configuration to optimize as secondary criteria (which we used throughout the remainder of the study), we conducted an empirical study to assess whether the tests became significantly less smelly. Furthermore, we studied how the proposed metrics affect the fault detection effectiveness, coverage, and size of the generated tests. Our study revealed that the proposed approach reduces the overall smelliness of the generated tests; in particular, the diffusion of the "Indirect Testing" and "Unrelated Assertions" smells improved considerably. Moreover, our approach improved the smelliness of the tests generated by EvoSuite without compromising the code coverage or fault detection effectiveness. The size and length of the generated tests were also not affected by the new secondary criteria.

Keywords: Test Smells, Software Quality, Automated Test Generation, Many-Objective Optimization, Empirical Studies

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This chapter presents the context, problem, and main motivations of our work.

1.1 Context

Software projects that strive to be successful must provide products that meet specific quality criteria [1, 15]. In this regard, software testing [64] is an essential process to ensure the production of high-quality software [45]; thus, it is also an invaluable practice in any successful software project [46]. However, testing is also quite an expensive process: in particular, manually written tests are very costly and time-consuming [22]. Even so, it is possible to reduce these costs through the automation of test creation [28]. Automated test generation tools, which support developers in testing activities [4, 46], have been successfully applied. For instance, the EvoSuite[1] tool has been able to detect real faults in financial programs [3] and achieve high code coverage on open-source and industrial software [19, 20, 38, 72, 93].

1.2 Problem Statement

Automated test generation tools such as EvoSuite [35] and Randoop [66] have become very effective at generating tests with high code coverage [45]. However, like manually written tests, automatically generated tests are susceptible to bad design/programming practices [46, 69], i.e., **test smells** [88]. The presence of test smells may:

- Negatively impact test code comprehensibility and maintainability [13, 14].
- Hinder test code effectiveness [85, 86].
- Make test code more prone to changes and faults [85].
- Make production code more fault-prone [85].

The negative impact that test smells might have on the quality of both test and production code demonstrates the importance/benefits of generating smell-free test code (i.e., test

[1] https://www.evosuite.org

code without bad programming practices). Although test smells are present in both manually written and automatically generated tests, these two types of tests are fundamentally different and, as such, are not affected in the same way (e.g., the diffuseness of test smells differs) [46, 69].

Manually written tests: Test smells usually arise due to poor design decisions made during the creation of test suites [87]. Test smells are often highly diffused in software systems [13] and, despite their negative impact, tend to persist for a long time [18, 87]. Furthermore, test smells are more likely to arise in large software systems [13]. Some types of test smells tend to co-occur together [13].

Automatically generated tests: Are often affected by test smells [46] because most automatic test generation tools mainly focus on producing tests with the highest possible code coverage and do not consider other factors which can be relevant for testers [70]: one such factor is the quality of the generated tests [69]. Therefore, test smells tend to be highly diffused throughout the automatically generated test suites due to the nature of these tools [46]. Moreover, other factors may influence the presence of specific smells in the generated tests, such as the quality of the production code and the size of the test suites [46]. Like manually written tests, automated tests also contain certain test smells that tend to be more diffused and that frequently co-occur together [69], but these differ between the two types of tests [46]. The presence of test smells in automatically generated tests is particularly concerning because developers need to understand the test code and infer its objective [84] before they can either add new assertions or analyze the existing automatically generated assertions to identify problems [35]. Typically, automatically generated test code is harder to understand [25] and maintain [84] than manually written code, and the presence of test smells further exacerbates this problem [13, 14]. Ultimately, the additional time required to perform maintenance tasks on automatically generated test code should not exceed the time saved through the automation of test creation [84]. Otherwise, the usefulness of these tools becomes debatable.

1.3 Approach

To the best of our knowledge, there are two main ways to avoid smelly tests:

1. Use specific refactoring operations to remove existing test smells [88]. Several test smell detection tools have been developed to assist in this process [2].
2. Generate tests that are smell-free by design, which is the objective of this work.

The primary goal of this project is to extend the EvoSuite tool such that the generated test code is smell-free (i.e., not affected by bad programming practices). We aim to achieve this objective without compromising the code coverage or fault detection effectiveness of the generated tests. For this purpose, we incorporated a carefully selected set of test smell metrics into EvoSuite and optimized these metrics as **secondary criteria** [45, 46, 70].

Indeed, during the search process, if multiple test cases are equally good in terms of coverage, then EvoSuite uses the secondary criteria to prioritize test cases with other desirable characteristics (e.g., number of statements). As such, by optimizing the test smell metrics as secondary criteria, tests are compared in terms of test smell metrics if and only if they cover the same code elements. To implement this approach, we need to replace the default secondary criterion (test case length) with new secondary criteria that optimize test smell metrics. We believe that the optimization of test smell metrics as secondary criteria corresponds to a promising way to generate less smelly tests, as similar approaches have already been successfully used to incorporate other non-coverage quality metrics into the EvoSuite tool [45, 70].

Overall, the underlying hypothesis of this study is the following:

> *We can optimize test smell metrics to address the high diffusion of test smells in the test code automatically generated by the EvoSuite tool. By optimizing a set of test smell metrics as secondary non-coverage-based criteria, we can promote less smelly tests with minimal impact on the final code coverage or fault detection effectiveness. Thus, it should be possible to improve the quality of the generated tests without compromising other aspects of the test code.*

1.4 Contributions

The main contributions of this thesis are as follows:

- We curated a list of 54 test smells, identified those that can affect the tests generated by EvoSuite, and established which of the identified test smells can be characterized by optimizable metrics.
- We investigated two different approaches to optimize test smell metrics in EvoSuite: (1) as secondary criteria; (2) as additional post-processing steps.
- We studied the diffusion of smells in the tests generated by EvoSuite; moreover, we conducted an empirical study to assess the test smells that affect a significant portion of the generated tests.
- We extended EvoSuite with new secondary criteria that optimize test smell metrics.
- We conducted an empirical study to assess the optimal combination of test smell metrics to optimize as secondary criteria.
- We conducted a large empirical study to assess whether the optimization of test smell metrics: (1) significantly reduces the smelliness of the generated tests; (2) does not negatively affect the coverage or fault detection effectiveness of the tests; (3) does not increase the number of generated tests.

The work developed in this thesis has been adapted into a research paper and presented at the 16th edition of the International Workshop on Search-Based and Fuzz Testing.

1.5 Structure of the Document

This document is organized as follows:

- Section 2 provides an overview of the background information required to understand the approaches we have proposed to optimize test smell metrics.
- Section 3 contains the related work, i.e., the literature associated with test smells.
- Section 4 describes how we identified the types of test smell metrics to optimize, how we intend to compute the respective metrics, and presents three approaches to optimize test smell metrics.
- Section 5 presents the design of the empirical study and describes the evaluation of the proposed approach.
- Section 6 discusses the results of the study.
- Section 7 summarizes this study and discusses additional future work.

Chapter 2

Background

This chapter presents background information about three different types of smells: code, architectural, and test smells. In addition, it also provides a general overview of different automatic test generation approaches and explains how the EvoSuite tool works (however, we will primarily focus on features related to the context of this work).

2.1 Code Smells

Code smells [33] correspond to suboptimal programming/design practices [67, 89]. These code smells equate to neither faults nor errors; instead, they indicate that the code might contain specific quality issues [95]. Indeed, code smells act as symptoms of possible problems [32] and, therefore, help to decide when/how to refactor [33]. After uncovering the presence of code smells in a given program, it is possible to apply certain refactoring operations to remove them [33] (code smell detection tools can aid in this process [31]). The presence of code smells can potentially:

- Hinder understandability and maintainability [95].
- Make the code more prone to changes [53, 68] and faults [67, 68].

Furthermore, these problems tend to become more severe when the same programs are affected by multiple smells [68]. Code smells are also a symptom of technical debt [10].

The following list enumerates some examples of code smells defined by Martin Fowler and Kent Beck in the book "Refactoring Improving the Design of Existing Code" [33]:

- **Duplicated Code:** Similar or equal code structures are present in several parts of a program [58]. Duplication may compromise the maintainability of systems [95]. Duplicated code can arise across [33]: (1) multiple methods of the same class; (2) sibling subclasses; (3) unrelated classes.
- **Long Method:** Excessively long methods (in terms of lines of code) that tend to encompass too many responsibilities [58]. This smell can make the code overly complex [32] and, therefore, difficult to understand, extend, and maintain [31].

- **Feature Envy:** Methods that access more data/functionality from classes other than their own [32, 33]. Methods affected by this smell might be in the wrong class [58].
- **Divergent Change:** Several parts of a specific class frequently need to be changed for various reasons [33]. This smell reveals a lack of cohesion in a class [58].
- **Shotgun Surgery:** The opposite of the "Divergent Change" code smell — a single change requires many small changes to multiple classes [33, 58].
- **Data Clumps:** Data items that always appear together — if an item is absent, the others may become meaningless [58]. Therefore, it is better to organize such data items into an object [33].
- **Lazy Class:** Classes that do not do enough to justify their existence; thus, they should be deleted [33]. Such classes are usually very small and simple [68].

The definitions of code smells are, however, subjective: for example, it is not possible to define the specific number of lines of code that indicates that refactoring should be applied in the "Long Method" code smell [89]. In some cases/projects, it might be *N* number of lines; in other cases, it might be *M*.

2.2 Architectural Smells

Architectural smells (AS) represent architectural decisions that compromise the quality of software systems in terms of maintainability and evolvability [11, 30]. Moreover, they are a source of technical debt [81] and, over time, can lead to architectural erosion [43]. Architectural smells suggest that there may be issues in the architecture [81].

Four examples of important types of architectural smells that have been described in the related literature [81]:

- **Unstable Dependency (UD):** A given component is dependent on other less stable components [11]. Such dependencies might trigger chains of changes [81].
- **Cyclic Dependency (CD):** Several components are directly/indirectly dependent on each other [11] (these dependent components form a cycle [81]).
- **Hub-like Dependency (HL):** Components have numerous incoming and outgoing dependencies [11, 81].
- **God Component (GC):** A component is much larger than the other components in the system [81] (i.e., it has too much control [11]).

Architectural smells can negatively affect the quality of software systems and, therefore, should be removed [43]. Hence, several architectural smell detection tools have been developed to help with this removal process [11].

2.3 Test Smells

As with production code, test code can also be affected by a particular set of bad smells that may jeopardize the quality of both test [13, 14, 85, 86] and production [85] code: **test smells** [63, 88]. Test smells correspond to suboptimal design/programming practices specific to test code that correlate with test implementation, organization, documentation, and interactions [14, 77, 87]. These smells are symptoms of possible problems in the test code and are often highly diffused in both manually written [13, 14] and automatically generated tests [46, 69]. Due to the high diffusion and potential dangers of test smells, researchers have [2]: (1) proposed new types of test smells; (2) developed test smell detection tools; (3) proposed refactoring operations to remove particular test smells.

This section depicts 54 test smells that, to the best of our knowledge, are representative of the most common types of test smells that have been proposed and evaluated in the related literature [2, 63]. Furthermore, we combine different descriptions/names for the same type of test smells into a single definition. If there are several similar types of test smells, then we only consider the most common of those smells; in turn, we regard the other test smells as variants. Whenever possible, we also provide an illustrative example for the test smells that are relevant to the context of this study.

1 – Abnormal UTF-Use (AUU) [2, 79]:

Description: Test suites change the default behavior of the unit-testing framework.
Impact: Harder to understand and maintain.

2 – Anonymous Test (AT) [2, 79]:

Description: Test cases with non-descriptive names.
Impact: Harder to understand and maintain.
Alternative Designations: Test-Method Category Name [79].

3 – Assertion Roulette (AR) [46, 63, 78, 86, 88]:

Description: A test case has several unexplained assertions. This test smell can arise for one of two reasons:

1. A test case has an excessive number of assertions (typically because the test case is inspecting too much functionality);
2. A test case has multiple assertions without assertion messages[1].

Impact: When the test fails, it is difficult to identify the exact assertion that failed (hinders comprehensibility and maintainability).

[1]An assertion message is an optional argument in an assertion statement that defines a message to be displayed when the assertion fails.

Listing 2.1: Example of a test case with the “Assertion Roulette” test smell [69].

```
@Test
public void test8() throws Throwable {
  Document document0 = new Document("", "");
  assertNotNull(document0);

  document0.procText.add((Character) 's');
  String string0 = document0.stringify();
  assertEquals("s", document0.stringify());
  assertNotNull(string0);
  assertEquals("s", string0);
}
```

4 – Brittle Assertion (BA) [2, 50]:

Description: One or more assertions in a test case check values that said test case does not manipulate (the test is checking too much).
Impact: Less effective (the test may fail when it should not) and harder to maintain.

5 – Conditional Test Logic (CTL) [2, 63, 77, 78, 86]:

Description: Test cases have control structures that may prevent the execution of specific statements (i.e., the test cases have many execution paths).
Impact: Behave unpredictably and are less effective (this smell may hamper test coverage and fault detection effectiveness). Such tests are also harder to understand and maintain.
Alternative Designations: Indented Test [16, 63] and Guarded Test [79].
Variants: Control Logic (ConL) [2, 79] — test cases use methods such as debug or halt to control the execution flow.

6 – Constructor Initialization (CI) [2, 77, 78, 92]:

Description: A test suite uses a constructor to initialize the fields; instead, tests should have set up methods (using constructors is a bad practice and, as such, should be avoided).
Impact: Harder to understand.

7 – Dead Field (DF) [2, 47]:

Description: Fields are initialized in an implicit setup but not used by the test cases.
Impact: This smell affects understandability and leads to slower tests (unnecessary work).
Alternative Designations: Unused Shared-Fixture Variables [79].

8 – Default Test (DT) [2, 77, 78]:

Description: Using the example test suites automatically generated by Android Studio.
Impact: If not removed, developers may add test cases to such test suites.

9 – Duplicate Assert (DA) [2, 77, 78, 92]:

Description: A test case contains several assertions that check the same condition (i.e., a test case contains two or more assertions with the same parameters). This smell arises even if the test case checks the same condition with different values.
Impact: Harder to understand and maintain.

Listing 2.2: Example of a test case with the "Duplicate Assert" test smell [77].

```
@Test
public void testXmlSanitizer() {
  .....
  valid = XmlSanitizer.isValid("Fritz-box");
  assertEquals("Minus is valid", true, valid);
  System.out.println("Minus test - passed");
  valid = XmlSanitizer.isValid ("Fritz-box");
  assertEquals("Minus is valid", true, valid);
  System.out.println("Minus test - passed");
  .....
}
```

10 – Eager Test (ET) [63, 74, 78, 86, 88]:

Description: A test case checks multiple methods of the class under test (i.e., it verifies too much functionality). Typically, this smell is said to occur when a test case checks two or more methods of the class under test.
Impact: Harder to understand and maintain.

Listing 2.3: Example of a test case with the "Eager Test" test smell [74].

```
@Test
public void test56() throws Throwable {
  SubstringLabeler substringLabeler0 = new SubstringLabeler();
  substringLabeler0.connectionNotification("testSet", "testSet");
  InstanceEvent instanceEvent0 = substringLabeler0.m_ie;
  substringLabeler0.acceptInstance(instanceEvent0);
  assertEquals("SubstringLabeler", substringLabeler0.getCustomName());
  assertFalse(substringLabeler0.isBusy());
  assertEquals("Match", substringLabeler0.getMatchAttributeName());
}
```

11 – Empty Shared-Fixture (ESF) [2, 79]:

Description: A test suite contains an implicit setup with an empty body.
Impact: Harder to understand and maintain.

12 – Empty Test (EmT) [2, 77, 78, 86, 92]:

Description: Test cases without executable statements.
Impact: Less effective (empty tests always pass, thus giving a false sense of security).
Variants: Comments Only Test (COT) [79] — corresponds to a commented-out test case.

13 – Erratic Test (ErT) [63]:

Description: Tests exhibit erratic behavior — they might give different results in different runs (e.g., different people run the same tests but obtain different results).
Impact: Less effective.

14 – Exception Handling (EH) [2, 77, 78]:

Description: Tests use throw/catch statements instead of JUnit's exception handling.
Impact: Harder to understand and maintain. This smell also makes tests less effective.

15 – For Testers Only (FTO) [13, 14, 46, 63, 88]:

Description: The production class contains code exclusively used by tests.
Impact: This test smell negatively impacts the understandability and maintainability of production code since it makes the system under test more complex.

16 – Fragile Test (FT) [63]:

Description: Tests start to fail to compile/run due to unrelated changes to the system under test. Fragile tests can also start to fail in situations where nothing was changed.
Impact: This smell leads to increased maintenance costs.

17 – Frequent Debugging (FD) [63]:

Description: It is often necessary to manually debug the tests to determine the cause of failures. This test smell might arise due to (1) the lack of available information or (2) infrequently run tests.
Impact: Such tests are less effective and harder to understand and maintain.
Alternative Designations: Manual Debugging [63].

18 – General Fixture (GF) [13, 47, 63, 86, 88]:

Description: An implicit setup is excessively general/large, so the test cases do not access the entirety of the fixture.
Impact: This smell affects comprehensibility and causes tests to run slower due to the unnecessary extra work. It can also make the tests fragile (i.e., affects maintainability).

19 – Hard-to-Test Code (HTTC) [63]:

Description: The system under test has properties that make it inherently difficult to test (e.g., highly coupled code). It is also possible to have test code that is difficult to test.
Impact: Less effective automatically generated tests. Harder to manually write tests.

20 – Ignored Test (IgT) [2, 77, 78, 86, 92]:

Description: Test case/suite uses the *@Ignore* annotation, thus impeding it from running.
Impact: Test suites become harder to understand. Causes compilation time overhead.

21 – Indirect Testing (IT) [13, 46, 63, 74, 88]:

Description: A test case performs tests on classes other than the one under test. This smell may arise because the test case is (indirectly) checking the respective production class using methods of other classes.
Impact: This smell negatively affects the comprehensibility and maintainability of test cases. It can also hamper the debugging process.

Listing 2.4: Example of a test case with the "Indirect Testing" test smell [69].

```
@Test
public void test3() throws Throwable {
  SweetHome3D sweetHome3D0 = new SweetHome3D();
  HomeRecorder.Type homeRecorder_Type0 = HomeRecorder.Type.DEFAULT;
  HomeFileRecorder homeFileRecorder0 = (HomeFileRecorder)
      sweetHome3D0.getHomeRecorder(homeRecorder_Type0);
  assertNotNull(homeFileRecorder0);
}
```

22 – Lack of Cohesion of Methods (LCM) [2, 47]:

Description: Unrelated test cases are arranged into a test suite (i.e., they are not cohesive).
Impact: Harder to understand and maintain.

Listing 2.5: Example of a test suite with the "Lack of Cohesion of Methods" test smell.

```
@Test
public void test00() throws Throwable {
  Login login0 = new Login();
  int int0 = login0.getAuth_num();
  assertEquals(0, login0.getAuth_max());
  assertEquals(0, int0);
}

@Test
public void test01() throws Throwable {
  int int0 = UserManagement.getNBGM();
  assertEquals(0, int0);
}
```

23 – Lazy Test (LT) [2, 13, 14, 78, 88]:

Description: Multiple test cases in a test suite check the same production method.
Impact: This smell can hinder maintainability.

Listing 2.6: Example of a test suite with the "Lazy Test" test smell [78].

```
@Test
public void testDecrypt() throws Exception {
```

```
  FileInputStream file = new FileInputStream(ENCRYPTED_DATA_FILE_4_14);
  byte[] enfileData = new byte[file.available()];
  FileInputStream input = new FileInputStream(DECRYPTED_DATA_FILE_4_14);
  byte[] fileData = new byte[input.available()];
  input.read(fileData);
  input.close();
  file.read(enfileData);
  file.close();
  String expectedResult = new String(fileData, "UTF-8");
  assertEquals("Testing simple decrypt",expectedResult,
      Cryptographer.decrypt(enfileData, "test"));
}

@Test
public void testEncrypt() throws Exception {
  String xml = readFileAsString(DECRYPTED_DATA_FILE_4_14);
  byte[] encrypted = Cryptographer.encrypt(xml, "test");
  String decrypt = Cryptographer.decrypt(encrypted, "test");
  assertEquals(xml, decrypt);
}
```

24 – Likely Ineffective Object-Comparison (LIOC) [2, 79]:

Description: A test case has one or more object comparisons that will never fail (e.g., a test case compares an object with itself).
Impact: Less effective.

Listing 2.7: Example of a test case with the "Likely Ineffective Object-Comparison" test smell [74].

```
@Test
public void test58() throws Throwable {
  OrganizationImpl organizationImpl0 = new OrganizationImpl();
  boolean boolean0 = organizationImpl0.equals(organizationImpl0);
  assertTrue(boolean0);
  assertEquals(0L, organizationImpl0.getPrimaryKey());
}
```

25 – Magic Number Test (MNT) [2, 77, 78, 86, 92]:

Description: A test case uses unexplained/undocumented numerical values.
Impact: This test smell hampers the understandability and maintainability of test cases (it is harder to understand the meaning and purpose of such values).

26 – Manual Intervention (MI) [63]:

Description: Tests that require some form of manual action to run.
Impact: Such tests are likely to be run less frequently due to the effort they require. As such, this smell makes tests less effective.

27 – Mixed Selectors (MS) [2, 79]:

Description: A class contains both production methods and test cases.

Impact: This test smell negatively impacts understandability and maintainability.

28 – Mystery Guest (MG) [13, 46, 63, 88, 92]:

Description: Tests use external resources (such as files or databases); hence, they are not self-contained.
Impact: Harder to understand and maintain due to the lack of available information. The usage of external resources also introduces hidden dependencies.

29 – Non-Java Smells (NJS):

Description: This does not correspond to a specific smell. Instead, it represents a set of simple test smells associated with concepts unrelated to Java. We have decided to combine the following smells into this category:

- **Empty Method Category [79]:** Test case with an empty method category.
- **Empty Test-Method Category [79]:** Test case with an empty test method category.
- **TTCN-3 Smells [12]:** Set of test smells specific to TTCN-3 test suites.
- **Unclassified Method Category [79]:** Test cases not organized by a method-category.

30 – Obscure In-line Setup (OISS) [2, 47]:

Description: A test case contains an excessive amount of setup functionality (an in-line setup should only have what is required to understand the test).
Impact: Harder to understand and maintain.
Variants: Max Instance Variables (MIV) [79] — Overly large fixture.
Note: The acceptable amount of setup information in a test case is dependent on the characteristics of the respective test and the production class.

31 – Overcommented Test (OCT) [2, 79]:

Description: A test contains too many comments.
Impact: This smell can make the tests harder to understand (which is the opposite of what comments should do).

32 – Overreferencing (OF) [2, 79]:

Description: A test case that references classes an excessive number of times.
Impact: This test smell makes the tests more difficult to understand and maintain.
Note: The acceptable amount of referenced classes in a test case is dependent on the characteristics of the respective test and the production class.

33 – Proper Organization (PO) [2, 79]:

Description: Poorly organized test cases that do not respect testing conventions.
Impact: Harder to understand and maintain.

34 – Redundant Assertion (RA) [2, 77, 78, 92]:

Description: Test cases have assertions that are permanently true/false (e.g., assertions with equal values for the actual and expected parameters).
Impact: This smell makes the tests less effective (it can give a false sense of security).

Listing 2.8: Example of a test case with the "Redundant Assertion" test smell [77].

```
@Test
public void testTrue() {
  /* ** Assert statement will always return true ** */
  assertEquals(true, true) ;
}
```

35 – Redundant Print (RP) [2, 77, 78]:

Description: Test cases have (unnecessary) print statements.
Impact: May hinder test effectiveness (print statements consume both time and resources).
Variants: Transcripting Test (TT) [79] — corresponds to printing/logging to the console.

36 – Resource Optimism (RO) [46, 78, 85, 86, 88]:

Description: Test cases that make optimistic assumptions about the existence/state of external resources.
Impact: This smell can lead to non-deterministic test results.

37 – Returning Assertion (ReA) [2]:

Description: A test case contains assertions and also returns a value.
Impact: This smell affects maintainability and comprehensibility.

38 – Rotten Green Tests (RGT) [2, 5, 27]:

Description: Test cases affected by this smell can pass without executing at least one assertion, thus giving a false sense of security.
Impact: Less effective.
Variants: Early Returning Test (ERT) — the test case does not execute certain assertions because it returns a value too early.

Listing 2.9: Example of a test case with the “Rotten Green Tests” test smell [5].

```
@Test
public void testLoggerContainsLogEntry(){
  Logger logger = new Logger();
  logger.log("log1");
  logger.log("log2");
  for (LogEntry logEntry : logger.getLogEntries()){
    assertTrue(logger.containsLogEntry(logEntry));
  }
}
```

39 – Sensitive Equality (SE) [46, 74, 78, 85, 88]:

Description: A test has assertions that perform equality checks using the *toString* method.
Impact: Test cases become dependent on (irrelevant) details of the String used in the comparison. Moreover, if the *toString* method for an object changes, the test starts failing.

Listing 2.10: Example of a test case with the “Sensitive Equality” test smell [74].

```
@Test
public void test62() throws Throwable {
  SubstringLabeler.Match substringLabeler_Match0 = new SubstringLabeler.Match();
  String string0 = substringLabeler_Match0.toString();
  assertEquals("Substring: [Atts: ]", string0);
}
```

40 – Sleepy Test (ST) [2, 77, 78]:

Description: Temporarily stopping the execution of a test case.
Impact: Less effective — pausing a thread can trigger unexpected results.

41 – Slow Tests (SloT) [63]:

Description: Tests take a long time to run. This smell can arise as a result of (1) poorly designed test code or (2) the characteristics of the system under test.
Impact: Slow tests are less effective and are likely to be run less frequently.

42 – Teardown Only Test (TOT) [2, 79]:

Description: Test suites that only specify teardown.
Impact: This smell affects maintainability.

43 – Test Code Duplication (TCD) [2, 13, 14, 63, 88]:

Description: Unwanted duplication in the test code. Test code duplication can be present amongst several tests or within the same test.
Impact: This smell affects maintainability and comprehensibility.
Alternative Designations: Duplicated Code [16].

44 – Test Logic in Production (TLP) [63]:

Description: Production code contains logic that should solely be exercised when testing.
Impact: This smell makes the system under test more complex and fault-prone (serious problems can arise when running the test-specific code in a production environment).

45 – Test Maverick (TM) [2, 47]:

Description: A test suite contains test cases independent of the existing implicit setup.
Impact: Run slower (unnecessary extra work). Harder to comprehend and maintain.

46 – Test Pollution (TP) [2, 48]:

Description: Dependent tests that can use (i.e., read/write) shared resources.
Impact: This test smell negatively impacts understandability and maintainability.

47 – Test Redundancy (TR) [2, 57]:

Description: One or more test cases can be removed without affecting the fault detection effectiveness of the test suite.
Impact: Redundant test cases only make the test suite harder to understand and maintain.

Listing 2.11: Example of a test suite with the "Test Redundancy" test smell.

```
@Test
public void test00() throws Throwable {
  int int0 = Login.getPASSWORDENC();
  assertEquals(2, int0);
}

@Test
public void test01() throws Throwable {
  int int0 = Login.getPASSWORDENC();
  assertEquals(2, int0);
}
```

48 – Test Run War (TRW) [2, 13, 14, 63, 88]:

Description: Tests allocate resources (e.g., temporary files) used by various people.
Impact: These tests may fail if different people run them simultaneously.

49 – Test-Class Name (TCN) [2, 79]:

Description: Test suites with non-descriptive names.
Impact: Harder to understand and maintain.

50 – Unknown Test (UT) [2, 74, 77, 78, 92]:

Description: Test cases without valid assertions or *@Test(expected)* annotations.
Impact: Less effective because there are no assertions to check whether the results are as expected. Harder to understand and maintain as there are no assertions to examine, thereby making it harder to deduce the purpose of the test cases (this is especially apparent when test cases have non-descriptive names).
Alternative Designations: Assertionless [16] and Assertionless Test [79].
Variants: Under-the-carpet Assertion (UCA) [79] — corresponds to commenting-out assertions in a test case (if only failing assertions are commented-out from the test case, then it is considered an Under-the-carpet failing Assertion (UCFA) [79] smell).

Listing 2.12: Example of a test case with the "Unknown Test" test smell [77].

```
@Test
public void hitGetPOICategoriesApi() throws Exception {
  POICategories poiCategories = apiClient.getPOICategories(16);
  for (POICategory category : poiCategories) {
    System.out.println(category.name() + ": " + category);
  }
}
```

51 – Unused Inputs (UI) [2, 50]:

Description: A test case has no assertions to check the particular values that said test case manipulates (the test case is checking too little).
Impact: Less effective (as it may be unable to reveal faults) and harder to maintain.

Listing 2.13: Example of a test case with the "Unused Inputs" test smell [50].

```
@Test
public void testGetRowKey() {
  DefaultKeyedValues2D d = new DefaultKeyedValues2D();

  d.addValue(new Double(1.0), "R1", "C1");
  d.addValue(new Double(1.0), "R2", "C1");
  assertEquals("R1", d.getRowKey(0));
  assertEquals("R2", d.getRowKey(1));
}
```

52 – Unusual Test Order (UTO) [2, 79]:

Description: A test that directly calls other tests.
Impact: Such tests may manifest erratic behavior (i.e., they are less effective).

53 – Vague Header Setup (VHS) [2, 47]:

Description: Fields are initialized in the class header rather than in an implicit setup.
Impact: Harder to understand and maintain.

54 – Verbose Test (VT) [16, 63, 79, 86, 92]:

Description: A test case contains an excessive number of statements (i.e., the test is unnecessarily long). As a result, the test code is neither clean nor simple.
Impact: The excessive number of statements makes the tests harder to understand and maintain. Such tests are also more likely to contain other types of test smells [46].
Alternative Designations: Complex Test [63], Long Test [79] and Obscure Test [63].
Note: The acceptable number of statements in a test case is dependent on the situation at hand (e.g., 20 statements may be too much in some situations and not enough in others).

2.4 Automatic Test Generation

This section describes three distinct approaches that enable the automatic generation of software tests.

2.4.1 Random-Based Software Testing

It is possible to generate tests by using random search-based algorithms. These algorithms create test cases by combining randomly generated statements/inputs [61]. This type of search has a fundamental limitation: it does not have a feedback mechanism — so it is not suitable for finding solutions that are difficult to reach (highly specific portions of the overall search space, i.e., the set of all possible solutions) [39, 60]. However, this is not the case for every tool that uses random testing: for instance, Randoop [66] is an automated test generation tool that uses "feedback-directed random testing" to avoid generating tests with illegal/redundant inputs [41].

2.4.2 Symbolic Execution

Symbolic Execution (SE) [17, 55] corresponds to a software testing technique that uses symbolic values instead of concrete input values [82] and assigns symbolic expressions to program variables [60, 62]. This technique is used to explore the different paths of a program: each path represents a set of possible executions and is associated with a logical formula that derives from the symbolic values [9]. Logical formulas are used to deduce concrete input values that explore as many execution paths as possible [17].

2.4.3 Search-Based Software Testing

Search-Based Software Testing (SBST) corresponds to the utilization of metaheuristic search techniques to automate testing tasks [61], i.e., it is a way to approach test creation as an optimization problem [49].

Metaheuristic search techniques use heuristics to propose solutions for combinatorial problems [60]. These techniques require some form of guidance to be able to sample from the search space in a controlled manner: this guidance is provided by feedback from problem-specific objective functions [61]. Objective functions evaluate solutions of the search in accordance with the search goals [60] and score better potential solutions to the problem with better objective values[2] [83]. Therefore, the search is guided towards more promising areas of the search space [60]. There are two main types of metaheuristics [61]:

- **Local search:** One solution in the local neighborhood (i.e., set of solutions that can be created by making a small change to the current solution) is considered at a time.
- **Global search:** Many solutions in the search space are sampled at a time.

Hill Climbing is a type of local search algorithm that starts by randomly choosing an initial solution from the search space [60]. The local neighborhood is analyzed, and a solution with a better objective value replaces the current solution [60]. The process of investigating the neighborhood and finding new solutions repeats until the neighborhood of the current solution contains no other solution with a better objective value [52]. The final solution is locally optimal [61], but it is not guaranteed to be globally optimal: the result is entirely dependent on a randomly obtained starting solution [60]. Simulated Annealing [56] partially mitigates this issue by having a given probability of replacing the current solution with a neighbor with a worse objective value: this is very likely to happen at the beginning of the search, but the probability decreases as the search progresses [60].

Genetic Algorithms (GAs) are a type of global search algorithm inspired by natural evolution (the fittest survive, evolve, and reproduce) [61]: a single solution corresponds to a chromosome/individual; a set of solutions constitutes a population; a population may evolve to form a new population, thus creating a new generation [60]. The initial population of candidate solutions is randomly generated [83]. To evolve the population, selection mechanisms are used to choose the ("parent") solutions that will be used to generate the "offspring" for the next generation [60]: solutions with a better fitness value are more likely to be selected [39]. A population evolves through the usage of genetics-inspired operations such as *mutation*, which randomly modifies a solution, and *crossover*, which combines genetic material from parent solutions to produce new offspring [19, 20]. New generations are created until either the solution is found, or the search budget (i.e., a limit to the resources that can be spent) is exhausted [61]. EvoSuite uses this type of search algorithm to automatically generate tests.

[2] A "better" solution simply refers to a solution that is closer to achieving the end goal of the search. Moreover, a "better" objective value does not necessarily imply that it has a greater score: in some instances, the search may be attempting to minimize the objective function and, therefore, the better solutions have lower values.

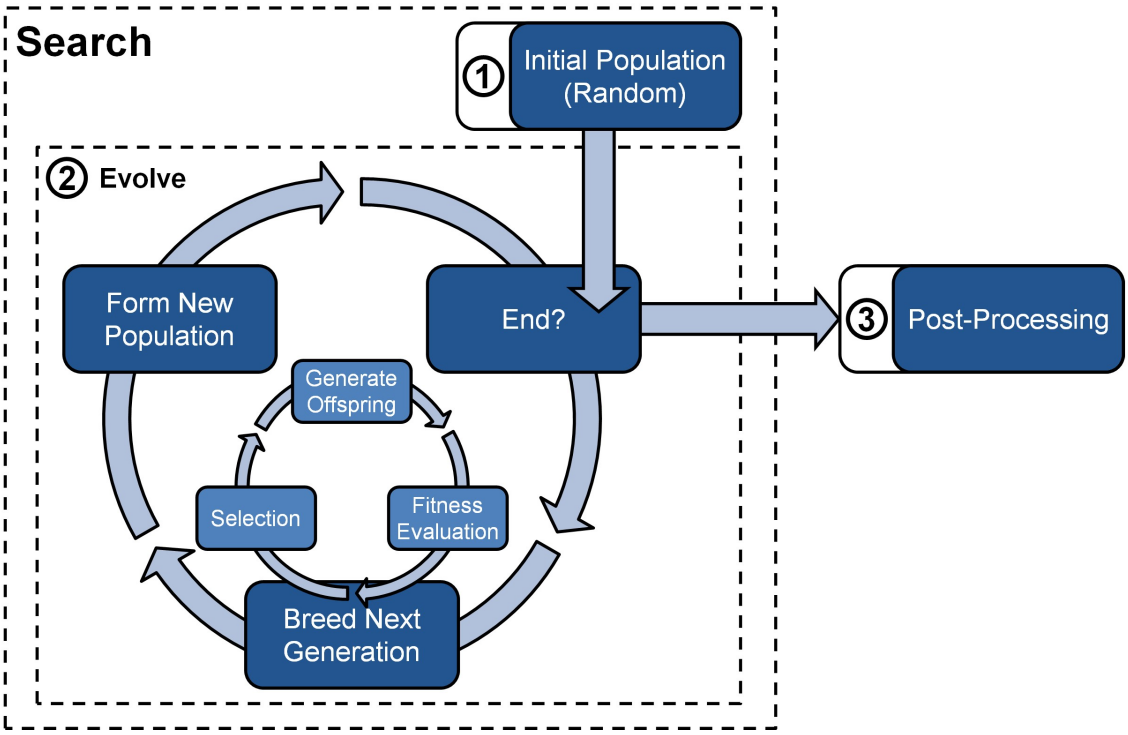

Figure 2.1: EvoSuite – Test generation process.

2.5 The EvoSuite Tool

EvoSuite [7, 34, 36, 93] is an automated search-based tool that uses evolutionary search to generate executable test suites for Java software. Given the class under test, the EvoSuite tool produces JUnit test cases that aim to maximize a specific set of coverage criteria [21]; indeed, EvoSuite is able to optimize several coverage criteria simultaneously. Moreover, the generated test cases contain regression assertions to capture the current behavior of the system under test [7].

By default, EvoSuite attempts to satisfy a specific range of testing criteria, but it is also possible to set other combinations of criteria [80]. Likewise, there also exists a set of properties that can be manipulated to configure how EvoSuite behaves [34]. The default search algorithm currently used by the EvoSuite tool is the Dynamic Many Objective Sorting Algorithm (i.e., DynaMOSA) [72], which operates at the test case level (test cases correspond to the chromosomes) [93]. DynaMOSA is an extension of MOSA [20] that uses a control dependency graph (CDG) to focus the search on targets (e.g., branches) that are free of control dependencies [72, 93].

Figure 2.1 depicts the test generation process presently used by the EvoSuite tool. This process can be divided into two main categories: (1) Search process and (2) Post-processing steps.

2.5.1 Search Process

The EvoSuite tool starts the search process by creating an initial population of randomly generated test cases — this initial population constitutes the first generation of tests. The population evolves over several generations in order to optimize a set of coverage criteria: EvoSuite compares tests and prioritizes those that lead to better code coverage (i.e., the fittest). Specifically, during each iteration of the evolutionary cycle, EvoSuite:

1. Generates new individuals (i.e., offspring population) from the current population (i.e., parent population);
2. Creates the population for the next generation by using test cases from the union of both parent and offspring solutions.

This evolutionary cycle continues until all the targets have been covered or the search budget has been fully exhausted. The end of the evolutionary cycle also marks the end of the search process.

EvoSuite uses a **secondary preference criterion** to promote test cases that are closer to uncovered targets and that have the shortest possible length — if multiple test cases have the same fitness value for a certain target, then the shortest one is preferred [72]. Test case length is used as the default secondary preference criterion because shorter test cases are easier to manually analyze [72] and less likely to break or expose flakiness [71].

2.5.2 Post-Processing

By default, after exhausting the search budget or achieving 100% code coverage, EvoSuite applies several post-processing steps to improve the quality/readability of the generated tests [34, 71, 93]: primitive values and null references are inlined, redundant test cases and statements (which do not contribute to the final code coverage) are removed, and a minimized set of assertions is added to each test case (using mutation analysis) [37]. The post-processing steps can be activated/deactivated by changing EvoSuite's properties.

2.6 Summary

In this chapter, we have presented three types of smells, i.e., code, architectural, and test smells. Given the context of this work, we primarily focused on test smells; specifically, we compiled test smells from several sources into a list of 54 smells (this list forms the basis of our work). After providing a general overview of different automatic test generation approaches, we described the main characteristics of the EvoSuite tool; in particular, we explained the: (1) search process; (2) secondary criteria; (3) post-processing steps. Having established both how EvoSuite automatically generates test cases and also a vast set of test smells, we can now proceed to investigate the integration of smell-free metrics in the EvoSuite tool.

Chapter 3

Related Work

Code smells were initially defined by Fowler [33] as structures in the code that suggest the possibility of refactoring, thus helping to decide when and how to refactor. Specifically, in the refactoring book [33], Beck and Fowler present a list of 22 code smells and the respective refactoring strategies to remove them.

Van Deursen et al. [88] extended the concept of code smells to test code and presented a catalogue of 11 test-specific smells (or test smells), along with refactoring operations to remove said smells. Meszaros [63] defined other test smells and considered test smells as the set of both code smells (i.e., code-level smells) and behavior smells (i.e., smells that affect the outcome of the test).

Bavota et al. [14] investigated the distribution of test smells and their impact on test code understandability and maintainability — the results revealed that: (1) test smells are highly prevalent in open-source and industrial software; (2) test smells can compromise test code understandability and maintainability; (3) specific test smells often co-occur together. Subsequently, Bavota et al. [13] carried out a more in-depth investigation into the same subject and confirmed the previously obtained results; furthermore, the researchers observed that larger software systems are more likely to be affected by test smells and that inexperienced developers are more susceptible to the effects of test smells than those with more experience. Note that the results of these two investigations are related to manually written tests, whereas we are focused on automatically generated tests (these two types of tests are fundamentally different). Bavota et al. [13, 14] also developed a tool that detects nine types of test smells in Java test suites. This tool uses rules that overestimate the smelliness of the tests, thus ensuring high recall at the expense of precision; indeed, this was a necessary design decision because the researchers simply intended to use this tool to alleviate the manual process of investigating the smelliness of software systems (and they did not want to miss any test-smell instances).

Spadini et al. [85] studied the correlation between six types of test smells and the quality of both test and production code. Regarding the impact of test smells on the quality of test code: (1) test smells make test cases more prone to changes and faults; (2) test

cases affected by multiple smells are more prone to changes than those affected by fewer smells; (3) tests containing the "Assertion Roulette", "Eager Test", and "Indirect Testing" smells are particularly more change- and fault-prone than those affected by the other considered test smells. Regarding the impact of test smells on the quality of production code: (1) production code exercised by smelly tests is more fault-prone than production code exercised by non-smelly tests; (2) the "Eager Test" and "Indirect Testing" smells have a stronger positive correlation with faulty production code than the other smells.

Tufano et al. [87] performed an empirical study to investigate: (1) developers' ability to recognize test smells; (2) the lifecycle of test smells. Through a survey, the researchers discovered that developers often fail to recognize test smells (and perceive their severity), thus demonstrating the importance of using automated test smell detection tools. Hence, to better understand the characteristics of test smells and therefore enable the development of automated detection tools, Tufano et al. investigated the lifecycle of test smells — the results demonstrated that: (1) test smells primarily occur due to the usage of bad design practices during the creation of test suites; (2) test smells tend to remain in software systems for a long time; (3) there may exist some correlation between code and test smells. In the case of our work, we do not intend to create a dedicated tool to detect test smells; instead, we seek to change the way EvoSuite generates tests to produce less smelly tests (even so, the depicted lack of awareness about test smells further proves the importance of extending the EvoSuite tool such that the generated test code is smell-free).

The negative influence of test smells has motivated the development of several test smell detection tools. Aljedaani et al. [2] presented a catalog of 22 test smell detection tools and compiled the smells detected by the respective tools into a single list of 66 test smells. We use this work as the primary source of test smells for our work (but we combine these smells with those presented by Meszaros [63]).

Peruma et al. [77] presented 12 new types of test smells (11 smells that apply to both Java and Android apps and one that is specific to Android apps) and demonstrated through a survey that the considered smells do indeed correspond to bad programming practices. Additionally, the authors investigated the distribution of test smells in Android applications. To this aim, the researchers decided to use tsDetect[1], an automated test smell detection tool for Java software that can detect 19 different types of test smells [76]. The results of the study revealed that test smells: (1) were highly diffused in the investigated test suites; (2) tended to be introduced early in the lifetime of the apps.

Virgínio et al. [92] investigated the association between test smells and code coverage and concluded that test smells might influence test code coverage. The authors used JNose Test[2], an automated test smell detection tool (with the ability to detect 21 different test smells) that also collects code coverage metrics. The JNose Test reuses the test smell

[1]https://github.com/TestSmells/TestSmellDetector
[2]https://github.com/arieslab/jnose

detection rules from the tsDetect tool [91]. Similarly, we also used the test smell detection rules from the tDetect tool as the main inspiration for our test smell metrics. However, we had to adapt these rules to the context of our work: instead of analyzing concrete tests, our test smell metrics analyze objects that represent the tests; in turn, we are dependent on the implementation of the respective classes.

Greiler et al. [47] presented six types of test smells related to fixture setup (five of which correspond to new types of test smells) and refactoring strategies to remove them. Moreover, the authors implemented a static analysis technique to detect test fixture smells in the TestHound tool (a tool that detects test fixture smells and proposes refactoring operations) and demonstrated that fixture-related test smells affect industrial projects.

Huo and Clause [50] created a technique based on dynamic tainting to automatically analyze test oracles and detect two types of test smells: "Brittle Assertions" (test cases check too much) and "Unused Inputs" (test cases check too little). The authors developed OraclePolish (a tool that implements the proposed technique) and demonstrated that this tool was effective at detecting both types of smells at a moderate cost.

Spadini et al. [86] studied severity thresholds for 11 types of test smells. Firstly, the authors utilized the tsDetect tool to establish new severity thresholds for nine types of test smells. The calibration results indicated that: (1) four out of nine test smells should be characterized by higher thresholds; (2) the other five test smells did not require higher thresholds. Subsequently, the authors investigated developers' perception of the 11 test smells considered in the study (using the established thresholds) — the researchers observed that: (1) the "Empty Test" and "Sleepy Test" smells had the highest refactoring priority; (2) the "Conditional Test Logic", "Empty Test", and "Ignored Test" smells had the most significant impact on test code maintainability; (3) the new severity thresholds better represented the developers' understanding of test smells; (4) even with the new thresholds, there were instances in which the developers still did not consider the detected test smells as real problems. Regarding our work, instead of verifying whether the tests are smelly, we optimize test smell metrics, i.e., we do not use thresholds (e.g., instead of checking whether a test case is too long, we compare equivalent tests and choose the one with fewer statements). However, we use thresholds to investigate the smelliness of the final test suites generated by EvoSuite. In fact, we use the thresholds defined by Spadini et al. [86] to measure the "Assertion Roulette", "Eager Test", and "Verbose Test" smells.

Palomba et al. [69] investigated the extent to which the tests automatically generated by EvoSuite are affected by test smells — the study revealed that: (1) as with manually written tests, test smells are also highly diffused throughout automatically generated tests; (2) among the studied test smells, the "Assertion Roulette", "Eager Test", and "Test Code Duplication" smells were the most diffused types of test smells in the generated test suites; (3) specific types of test smells often co-occur together in automatically generated tests; (4) there exists a positive correlation between test smells and the structural properties of

the system. Subsequently, Grano et al. [46] extended upon the prior analysis [69] by studying the smelliness of the test suites automatically generated by three state-of-the-art tools: EvoSuite, Randoop, and JTExpert. The study revealed that: (1) the three tools generate smelly test code; (2) the "Assertion Roulette" and "Eager Test" smells constitute the most diffused types of bad smells in the tests generated by the three tools; (3) the generated tests are smelly due to the nature of the tools; (4) the presence of specific test smells may imply the presence of other test smells; (5) the size of the tests is associated to the occurrence of specific test smells. We extend these Palomba et al. [69] and Grano et al. [46] works as follows: (1) we perform our study on a newer version of the EvoSuite tool; (2) we consider a larger set of 16 smells and implement the respective test smell metrics; (3) instead of just detecting smells, we also optimize the proposed test smell metrics to generate less smelly tests.

Panichella et al. [74] conducted a study to determine the effectiveness of test smell detection tools at identifying smells in automatically generated test code. They used two test smell detection tools to detect six smell types in the test suites generated by the EvoSuite tool: the tool developed by Bavota et al. [13, 14] and the tsDetect tool [76]. Firstly, Panichella et al. performed a manual investigation to assess the smelliness of 100 test suites automatically generated by EvoSuite and observed that: (1) automatically generated tests are affected by a small but non-trivial quantity of test smells; (2) the "Assertion Roulette" and "Eager Test" smells frequently co-occurred together; (3) the "Indirect Testing" was the most diffused type of smell in the generated tests. Secondly, they compared the manually identified test smells with the smells reported by the two selected tools and concluded that: (1) both tools overestimated the smelliness of the generated tests; (2) test smell detection tools should use better metrics and detection strategies. Moreover, Panichella et al. stated that EvoSuite averts the "Mystery Guest" and "Resource Optimism" smells through the usage of mocks and bytecode instrumentation. The authors also presented three types of relevant problems not covered by the studied test smells. We also investigate the same six types of test smells considered in this study in our work; however, not only do we study other smells, but we also implement new test smell metrics (taking into account the feedback provided by the authors in this study). Subsequently, Panichella et al. [75] extended upon this work by investigating the same test smell detection tools and considering: (1) the tests generated by the EvoSuite and JTExpert tools; (2) manually written tests. Regarding automatically generated tests, the authors confirmed the results obtained in the previous study. Concerning manually written tests, the authors observed that: (1) manually written tests and automatically generated tests are affected by test smells differently; (2) both tools are more effective at identifying smells in manually written tests; (3) most detected smells did not correspond to problems.

Chapter 4

Approach

The main objective of this study is to integrate test smell metrics into the EvoSuite tool and optimize these metrics such that the generated test code is smell-free. We aim to achieve this goal without compromising the final code coverage or fault detection effectiveness of the generated tests.

We decided to implement these metrics into the EvoSuite tool because: (1) it is a popular state-of-the-art automated test generation tool that has achieved the highest score in various editions of the SBST tool competition [21, 42, 71, 93]; (2) has been successfully augmented with a multitude of other quality metrics [24, 25, 45, 70]; (3) several studies have investigated the presence of test smells in the generated tests and proposed potential solutions [46, 69, 74].

The test suites automatically generated by EvoSuite are affected by bad programming practices that hinder the quality of both test and production code: test smells [46, 69, 74]. These bad smells occur because the primary focus of the EvoSuite tool is to produce tests with the highest possible coverage; as such, certain aspects related to code quality (like the usage of good programming practices) are not adequately accounted for [69]. Moreover, as stated by Grano et al. [46], test smells arise due to the very nature of the EvoSuite tool: the initial population of tests is randomly generated, so it is only natural that they do not follow good programming practices, i.e., they are smelly. The population evolves over several generations, and the only way in which the smelliness of the tests is (to some extent) regulated is through the default secondary criterion, which prioritizes shorter tests. Even so, simply prioritizing smaller tests is not enough to ensure the absence of test smells; as such, once the tests become smelly, they tend to remain smelly throughout the evolutionary process. After the search, EvoSuite applies multiple post-processing steps to improve the quality of the tests, thus making the generated tests (potentially) less smelly [93]; however, different smells have different causes, and the post-processing steps do not address all types of test smells.

The tests generated by EvoSuite are indeed smelly, but, as described in Section 2.5, this tool also provides several ways to optimize quality aspects. Indeed, if we establish

metrics that characterize the test smells that we want to avoid, then we can implement these metrics into EvoSuite and take advantage of the different ways in which EvoSuite already optimizes quality-related aspects to optimize the test smell metrics and, as such, generate less smelly tests. To optimize test smell metrics in EvoSuite, we decided to:

1. Select the test smells for which we will define metrics (i.e., the metrics that we will consider to optimize). This process requires us to:
   (a) Establish which of the 54 identified test smells (listed in Section 2.3) can affect the tests generated by EvoSuite;
   (b) Select the test smells that we can characterize with optimizable metrics.
2. Define specific metrics for the selected test smells.
3. Identify viable ways to optimize the test smell metrics.

Table 4.1 depicts all the test smells considered in this study. We use this table to summarize the test smell selection process and represent the test smells with metrics that can be optimized as secondary criteria and/or as additional post-processing steps:

- The "Occur" column identifies which test smells can occur in the tests generated by the EvoSuite tool. The "Optimize" column identifies which of the test smells that can arise in the generated tests can be characterized by optimizable metrics. These two columns summarize the test smell selection process.
- The "Sec. Criteria" column identifies the test smells with metrics that can be optimized as secondary criteria. The "Post-Proc." column identifies the smells with metrics that can be optimized as additional post-processing steps. These two columns represent the two viable optimization approaches that we have identified.

4.1 Test Smell Selection

We must first establish which of the 54 identified test smells (see in Table 4.1) are relevant to the context of this work — these are the smells we want to select so that we may create and optimize the respective metrics and, consequently, improve the quality of the tests generated by EvoSuite. Specifically, we intend to select test smells that:

1. Can actually arise in the tests generated by EvoSuite.
2. Can be characterized by optimizable metrics.

Hence, to identify the relevant test smells, we: (1) assumed that all considered test smells could occur in the generated tests; (2) carefully analyzed the description of each test smell; (3) excluded the test smells that could not arise due to the characteristics of the generated tests; (4) excluded test smells that could not be characterized by optimizable metrics. We complemented our findings with the results of prior studies that had already investigated the presence of test smells in the tests generated by EvoSuite [46, 69, 74].

Table 4.1: Test smells considered in this study. The rows highlighted in gray correspond to the test smells with metrics that we will optimize as secondary criteria. Description of each column can be found at the end of Section 4.

ID	Name	Abbr.	Occur	Optimize	Sec. Criteria	Post-Proc.
1	Abnormal UTF-Use	AUU	NO	—	—	—
2	Anonymous Test	AT	YES	NO	—	—
3	Assertion Roulette	AR	YES	YES	NO	YES
4	Brittle Assertion	BA	YES	NO	—	—
5	Conditional Test Logic	CTL	NO	—	—	—
6	Constructor Initialization	CI	NO	—	—	—
7	Dead Field	DF	NO	—	—	—
8	Default Test	DT	NO	—	—	—
9	Duplicate Assert	DA	YES	YES	NO	YES
10	Eager Test	ET	YES	YES	YES	YES
11	Empty Shared-Fixture	ESF	NO	—	—	—
12	Empty Test	EmT	NO	—	—	—
13	Erratic Test	ErT	YES	NO	—	—
14	Exception Handling	EH	NO	—	—	—
15	For Testers Only	FTO	NO	—	—	—
16	Fragile Test	FT	YES	NO	—	—
17	Frequent Debugging	FD	YES	NO	—	—
18	General Fixture	GF	NO	—	—	—
19	Hard-to-Test Code	HTTC	NO	—	—	—
20	Ignored Test	IgT	NO	—	—	—
21	Indirect Testing	IT	YES	YES	YES	YES
22	Lack of Cohesion of Methods	LCM	YES	YES	NO	YES
23	Lazy Test	LT	YES	YES	NO	YES
24	Likely Ineffective Object-Comparison	LIOC	YES	YES	YES	YES
25	Magic Number Test	MNT	NO	—	—	—
26	Manual Intervention	MI	YES	NO	—	—
27	Mixed Selectors	MS	YES	NO	—	—
28	Mystery Guest	MG	NO	—	—	—
29	Non-Java Smells	NJS	NO	—	—	—
30	Obscure In-line Setup	OISS	YES	YES	YES	YES
31	Overcommented Test	OCT	NO	—	—	—
32	Overreferencing	OF	YES	YES	YES	YES
33	Proper Organization	PO	YES	NO	—	—
34	Redundant Assertion	RA	YES	YES	NO	YES
35	Redundant Print	RP	NO	—	—	—
36	Resource Optimism	RO	NO	—	—	—
37	Returning Assertion	ReA	NO	—	—	—
38	Rotten Green Tests	RGT	YES	YES	YES	YES
39	Sensitive Equality	SE	YES	YES	NO	YES
40	Sleepy Test	ST	NO	—	—	—
41	Slow Tests	SloT	YES	NO	—	—
42	Teardown Only Test	TOT	NO	—	—	—
43	Test Code Duplication	TCD	YES	NO	—	—
44	Test Logic in Production	TLP	NO	—	—	—
45	Test Maverick	TM	NO	—	—	—
46	Test Pollution	TP	NO	—	—	—
47	Test Redundancy	TR	YES	YES	NO	YES
48	Test Run War	TRW	NO	—	—	—
49	Test-Class Name	TCN	YES	NO	—	—
50	Unknown Test	UT	YES	YES	NO	YES
51	Unused Inputs	UI	YES	YES	NO	YES
52	Unusual Test Order	UTO	YES	NO	—	—
53	Vague Header Setup	VHS	NO	—	—	—
54	Verbose Test	VT	YES	YES	YES	YES

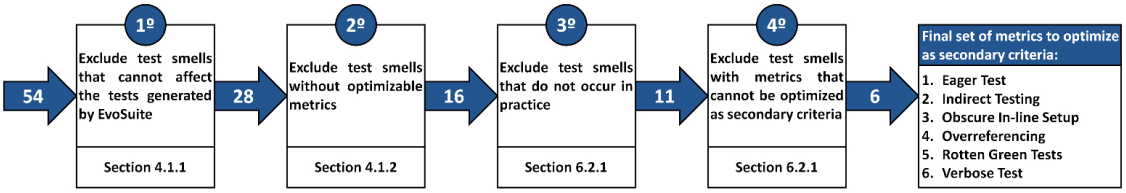

Figure 4.1: Test smell exclusion process. The four test smell exclusion steps that we performed to establish which of the 54 identified test smells we should combine in different ways to find the optimal combination of metrics to optimize as secondary criteria.

4.1.1 Test Smells That Cannot Occur

Some test smells cannot arise due to the characteristics of the tests generated by EvoSuite. Indeed, EvoSuite simply does not contain the code necessary to generate tests with certain smells. For instance, EvoSuite is unable to generate tests with conditional statements. As such, we decided to exclude test smells related to:

Implicit setups: Dead Field; Empty Shared-Fixture; General Fixture; Test Maverick. The test suites generated by EvoSuite do not use implicit setups (i.e., setup methods used by all the test cases in the test suite). Each test case contains the necessary setup code; hence, these test smells cannot arise in the generated tests. EvoSuite does generate scaffolding files that do the necessary setup/pulldown, but these are only used to avoid flaky tests and are not part of the scope of this work.

Teardown: Teardown Only Test. Given that the test suites generated by EvoSuite do not use setups, they also do not use teardowns.

Improper setup that is not contained in a test case: Constructor Initialization; Vague Header Setup. Neither bad smells occur because the test cases generated by EvoSuite always contain all the necessary setup code.

Problems that do not apply to JUnit tests: Default Test; Non-Java Smells; Returning Assertion. As these smells are not related to JUnit tests, they cannot be detected.

Non-existent statement types: Conditional Test Logic; Redundant Print; Sleepy Test. EvoSuite generates test suites that are composed of test cases. A test case corresponds to a sequence of statements. Each statement has a specific type that defines it. Therefore, it is impossible to generate test code for a statement type that does not exist.

Problems with the code under test: For Testers Only; Hard-to-Test Code; Test Logic in Production. These smells are not related to the generated test code. It would be necessary to analyze the source code to optimize the respective test smell metrics.

Annotations: Abnormal UTF-Use; Exception Handling; Ignored Test. The generated test cases are annotated only with *@Test*. The Exception Handling smell is inevitable because try/catch has to be used to deal with exceptions (i.e., it is necessary).

Comments: Overcommented Test. EvoSuite only adds comments to generated tests if necessary (e.g., when a try-catch block is used).

Inline: Magic Number Test. One of the post-processing steps provided by EvoSuite is used to inline all the primitive values and null references in the generated test cases. As such, the manifestation of this smell is dependent on whether the person that uses EvoSuite decides to apply (or not) this post-processing step. Therefore, we assume that this smell can only occur if the person using EvoSuite makes the conscious decision to disable the inline post-processing step.

Empty test cases: Empty Test. The minimization post-processing step provided by EvoSuite already removes all empty test cases in the test suites.

Use of external resources: Mystery Guest; Resource Optimism; Test Pollution; Test Run War. As stated by Panichella et al. [74], EvoSuite avoids test smells related to external resources through the usage of mocks and bytecode instrumentation.

Given the initial set of 54 test smells, we have identified 26 smells that definitely cannot arise in tests generated by EvoSuite. Hence, by excluding these smells, we obtain a subset of 28 test smells that may affect the generated tests. However, we still need to identify which of these 28 test smells can be characterized by optimizable metrics.

The "Occur" column of Table 4.1 presents the smells that can and cannot occur in tests generated by EvoSuite. The exclusion process of the test smells that cannot arise due to the characteristics of the generated tests is represented in the 1st step of Figure 4.1.

4.1.2 Test Smells Without Optimizable Metrics

We have a set of 28 test smells that may arise in the tests generated by EvoSuite. Still, before we implement the respective test smell metrics, we have to ponder whether it would even be possible to define optimizable metrics that characterize these smells. Indeed, some test smells can arise in the tests generated by EvoSuite, but the respective metrics cannot be optimized due to technical limitations. For instance, we cannot optimize metrics related to the code under test. Therefore, to identify the test smells that we should remove, we use the descriptions provided in Section 2.3 to deduce hypothetical metrics for each of the 28 remaining smells. Hence, we excluded 12 test smells:

Smells that cannot be automatically detected: Brittle Assertion; Erratic Test; Fragile Test; Frequent Debugging; Manual Intervention; Mixed Selectors; Proper Organization; Unusual Test Order. While these test smells can occur, they cannot be automatically detected: (1) for Brittle Assertion, we would have to identify all the values of the class under test influenced by test cases; (2) for Erratic Test, it would be necessary to run the same tests in different contexts; (3) for Fragile Test, it would be necessary to make changes to the code under test; (4) for Frequent Debugging, it is hard to deduce if the cause of a failure is unintuitive; (5) for Manual Intervention, we cannot verify whether the tests require some form of manual action; (6) for Mixed Selectors and Unusual Test Order, it would be necessary to distinguish production and test code; (7) for Proper Organization, it would be hard to create an optimization strategy for such a subjective

concept.

Meaningless/unclear names: Anonymous Test; Test-Class Name. The only way to optimize metrics related to such smells is to change the way EvoSuite creates these names in the first place.

Resource intensive metrics: Slow Tests; Test Code Duplication. These two test smells require metrics that are too resource-intensive: (1) for Slow Tests, it would be necessary to run the same test multiple times to calculate the respective average[1]; (2) for Test Code Duplication, it would be necessary to use a metric such as the Levenshtein Distance to check whether there are repeated groups of similar statements in a test case.

Given the initial set of 28 test smells, we have identified 12 smells that we definitely cannot characterize with optimizable metrics. Thus, by excluding these smells, we have obtained the final subset of 16 test smells for which we can and will define metrics (see Section 4.2). However, we still need to understand to what extent the tests generated by EvoSuite are affected by test smells. Indeed, we might be able to detect and optimize the metrics of the non-excluded test smells, but that does not necessarily imply that these smells affect the generated tests. Moreover, we do not want to spend resources attempting to optimize smells that never arise in the tests generated by EvoSuite. Therefore, after implementing the metrics for the non-excluded test smells into EvoSuite, we will analyze the diffusion of each test smell and verify whether it should be optimized. Once we have established the set of smells that arise in practice, we will attempt to optimize the respective metrics.

The "Optimize" column of Table 4.1 establishes which of the test smells that can arise in the generated tests can be characterized by optimizable metrics. The exclusion process of the test smells that cannot be characterized by optimizable metrics is represented in the 2nd step of Figure 4.1.

4.2 Test Smell Metrics

This section presents the set of 16 test smells that we can describe with optimizable metrics. For each test smell, we establish:

- **Adaptation (optional):** How we adapted the original definition of the smell to suit the context of this work;
- **Metric:** Description of the test smell metric;
- **Computation:** How to compute the test smell metric (present in the link to the source code);
- **Threshold:** Threshold that we use to investigate the smelliness of the generated tests (note that we do not utilize this threshold to perform the optimization).

[1]We have to run the same tests multiple times because the duration of the last execution of a test case (that EvoSuite calculates by default) is somewhat unreliable and unpredictable.

There are two main types of test smells:

1. **Test case smells (TCS):** Evaluated at the test case level;
2. **Test suite smells (TSS):** Evaluated at the test suite level.

To calculate the smelliness of test case smells, we simply have to analyze a specific test case. However, to compute the smelliness of test suite smells, we must have access to the entire test suite and analyze each of the test cases.

Assertion Roulette (TCS):

Adaptation: EvoSuite does not generate assertion messages, so this metric only focuses on avoiding an excessive number of assertions. A test case is only affected by this smell if the total number of assertions is greater than the total number of method calls.
Metric: Number of assertions in a test case that exceed the total amount of statements that call methods of the class under test.
Computation: available in *here*.
Threshold: 3 (defined by Spadini et al. [86]).

Duplicate Assert (TCS):

Adaptation: Two assertions are equal if they: (1) check the same method of the same class; (2) correspond to the same type of assertion; (3) have the same expected value.
Metric: Number of assertion statements of the same type that check the same method of the same class and have the same expected value.
Computation: available in *here*.
Threshold: 1 (defined by Peruma et al. [77, 78]).

Eager Test (TCS):

Metric: Total number of different methods of the class under test that are checked by a test case.
Computation: available in *here*.
Threshold: 4 (defined by Spadini et al. [86]).

Indirect Testing (TCS):

Metric: Total number of methods of other classes that are checked by a test case.
Computation: available in *here*.
Threshold: 1 (defined by Bavota et al. [13, 14]).

Lack Of Cohesion Of Methods (TSS):

Adaptation: A test suite has a specific target class: the class under test. All test cases in a test suite are supposed to perform tests on the class under test and, in that sense, should also be related. Therefore, instead of verifying if all test cases perform tests on a common class, in this context, a test case is considered smelly if it does not perform tests on the class under test.
Metric: Number of test cases in a test suite that do not perform tests on the class under test.
Computation: available in *here*.
Threshold: 1 (regarding the tests generated by EvoSuite, test cases that do not contribute to satisfying the coverage goals serve no purpose; therefore, a test case that does not even exercise the class under test will surely be redundant, and the final test suite should never contain redundant test cases).

Lazy Test (TSS):

Metric: Number of times each production method is called by more than one test case.
Computation: available in *here*.
Threshold: 1 (defined by Bavota et al. [13, 14] and Peruma et al. [77, 78]).

Likely Ineffective Object-Comparison (TCS):

Metric: Number of times the "equals" method of a class other than the one under test is used to compare an object with itself.
Computation: available in *here*.
Threshold: 1 (it only makes sense to use the "equals" method to compare an object with itself if "equals" corresponds to a method of the class under test; therefore, such a comparison should never be performed in other circumstances).

Obscure In-line Setup (TCS):

Adaptation: This metric does not consider the variables that store values returned from methods of the class under test.
Metric: Number of declared variables in a test case.
Computation: available in *here*.
Threshold: 10 (defined by Greiler et al. [47]).

Overreferencing (TCS):

Metric: Number of unnecessary class instances (i.e., class instances that are created but never used).

Computation: available in *here*.
Threshold: 1 (every object created in a test case should have a given purpose and, as such, should be used at least once).

Redundant Assertion (TCS):

Adaptation: EvoSuite never generates assertions with the same values for the actual and expected parameters. As such, this metric focuses on another type of redundant assertions: assertions that check primitive statements.
Metric: Total number of assertions that check primitive statements.
Computation: available in *here*.
Threshold: 1 (defined by Peruma et al. [77, 78]).

Rotten Green Tests (TCS):

Adaptation: This smell occurs if a test case continues to exercise code after the statement in which the first exception was raised.
Metric: Number of statements that exist after the statement that raised the first exception.
Computation: available in *here*.
Threshold: 1 (any code that exists after the first statement that raises an exception will not be executed and, as such, should be removed).

Test Redundancy (TSS):

Metric: Number of test cases that can be removed from the test suite without decreasing the code coverage.
Computation: available in *here*.
Threshold: 1 (regarding the tests generated by EvoSuite, test cases that do not contribute to satisfying the coverage goals serve no purpose; therefore, the final test suite should never contain redundant test cases).

Unknown Test (TCS):

Metric: Number of valid assertions in a test case.
Computation: available in *here*.
Threshold: 1 (defined by Peruma et al. [77, 78]).

Unrelated Assertions (TCS):

Adaptation: This test smell corresponds to an adaptation of the test smell "Sensitive Equality". We initially considered that the test smell "Sensitive Equality" should not be evaluated when an assertion performs an equality check using the toString method

implemented in the class under test. We then realized that the real problem with this new approach is not that there are assertions that check the toString method but that there are assertions that check methods not declared in the class under test. As such, we decided that this metric should instead consider all assertions which check methods not declared in the class under test. We changed the name of this test smell metric because this newly proposed metric diverges a lot from the original definition.
Metric: Total number of assertions that check methods that are not declared in the class under test.
Computation: available in *here*.
Threshold: 1 (assertions should capture the current behavior of the system under test; thus, assertions that check methods not declared in the class under test serve no purpose).

Unused Inputs (TCS):

Adaptation: Without having full access to the class under test, it is difficult to know for sure if an assertion does not check values controlled by the test case. Thus, the proposed metric only focuses on statements that should necessarily have assertions. Specifically, every statement which calls a method of the class under test that returns a value should necessarily have at least one assertion; otherwise, the test is considered smelly.
Metric: Number of assertionless statements that call methods of the class under test.
Computation: available in *here*.
Threshold: 1 (if a statement calls a method of the class under test that returns a value, then that statement should necessarily have an assertion to capture the current behavior of the system under test).

Verbose Test (TCS):

Metric: Total number of statements in a test case.
Computation: available in *here*.
Threshold: 13 (defined by Spadini et al. [86]).

4.3 Optimize Test Smell Metrics

Although the primary objective of EvoSuite is to maximize coverage, it is also possible to consider non-coverage aspects to improve the usefulness of the generated tests [70]. Through the analysis of previous studies, we were able to identify three possible approaches to incorporate non-coverage quality metrics into the EvoSuite tool:

1. Additional criteria [29, 59].
2. Secondary criteria [45, 70].
3. Additional post-processing steps [24, 25].

In theory, these three approaches should allow us to optimize test smell metrics. Thus, in this section, we investigate the advantages/disadvantages of each approach and study their practical feasibility.

4.3.1 Optimize Test Smell Metrics as Additional Criteria

The EvoSuite tool can optimize multiple coverage criteria simultaneously [71]. Therefore, it should also be possible to optimize the proposed metrics by regarding the test smell metrics as additional objectives to consider in addition to code coverage [70].

As demonstrated by Rojas et al. [80], it is possible for EvoSuite to simultaneously optimize multiple criteria without increasing the computational costs. In fact, by default, the current version of EvoSuite combines different coverage criteria to generate tests with higher code coverage [20, 73] and improved fault detection effectiveness [44]. However, several studies have shown that the combination of coverage and non-coverage-based objectives negatively impacts the final code coverage, as they tend to conflict with each other [29, 45, 59, 70].

The optimization of test smell metrics and the maximization of code coverage are conflicting goals: it would be possible for tests with worse coverage to be chosen because they are less smelly, which is not desirable. It should be theoretically possible to produce less smelly tests with this approach. However, as a result, we foresee that there would be a significant negative impact on code coverage. Therefore, this approach is not viable.

4.3.2 Optimize Test Smell Metrics as Secondary Criteria

During the search procedure, EvoSuite uses, by default, a secondary non-coverage-based criterion (i.e., test case length) that promotes test cases that have the shortest possible length [72]. In a nutshell, when there are multiple test cases with the same fitness value for a given coverage target, EvoSuite selects the shortest one. Researchers have explored this functionality and have successfully incorporated non-coverage metrics into EvoSuite as secondary criteria.

Palomba et al. [70] proposed a secondary criterion to generate more cohesive and less coupled tests, which combines two metrics: cohesion (LCTM) and coupling (CBTM). They evaluated their approach and concluded that it: (1) achieved better LCTM and CBTM scores relative to both MOSA and the whole suite strategy; (2) generated shorter tests; (3) achieved higher code coverage due to the lower probability of early convergence. Their study has demonstrated the viability of incorporating quality metrics into EvoSuite through the usage of a secondary criterion. Moreover, it revealed that adequate quality metrics could have a wide range of indirect benefits in the final test code.

Grano et al. [45] presented an adaptive search algorithm: aDynaMOSA — which extends DynaMOSA with performance proxies that estimate test execution costs (runtime

and memory consumption) as secondary criteria. Furthermore, aDynaMOSA's adaptive strategy is capable of temporarily disabling the secondary criteria when adverse effects on code coverage are identified. According to their results, it was possible to generate less expensive tests (which have decreased runtime and memory consumption) and also achieve code coverage and mutation score comparable to DynaMOSA.

Thus, we hypothesize that it might be possible to use secondary criteria to optimize quality aspects as test smells with minimal impact (if any) on the final code coverage [45]. In particular, test cases are compared in terms of test smell metrics if and only if they cover the same code elements (e.g., lines, branches, etc.), as opposed to the approach described in Section 4.3.1. Note that these newly proposed secondary criteria should also be able to keep the size of the test cases under control (as EvoSuite's default secondary criterion), as several types of test smells are directly associated with the size of the test cases [46]. For these reasons, we conclude that this approach is viable. Nevertheless, when it comes to implementing this approach, there are a few pitfalls, which we describe in detail below.

Pitfall 1 – Metrics that cannot be directly optimized as secondary criteria

The only test smell metrics that can be optimized as secondary criteria are those that are possible to compute during the search-based test generation process, i.e., some test smell metrics simply cannot be optimized as secondary criteria. However, as demonstrated by Grano et al. [46], the presence of specific test smells may imply the presence of other test smells. Therefore, it is possible for test smell metrics that cannot be computed during the test generation process to be indirectly optimized.

Pitfall 2 – More or less coverage

This approach could also negatively or positively affect the coverage achieved by the generated tests. Incidentally, there is evidence to suggest that code coverage might either decrease or increase due to "Genetic Drift" [54]. Genetic drift arises when the individuals of a population become too similar, thus diminishing the ability to explore the search space. Panichella et al. [72] demonstrated that this problem arises even when using test case length as the secondary criterion:

- On the one hand, the generated tests (i.e., the final test suite) should be as small and simple as possible.
- On the other hand, during the search process, complex and large tests have higher evolutionary potential, i.e., they are more capable of change and, as such, promote diversity. Consequently, by prioritizing shorter tests, the default secondary criterion compromises the capability to explore the search space.

For example, as stated by Palomba et al. [70], while redundancy decreases test readability, it helps to explore the search space; therefore, the optimization of the test smell metric

"Test Code Duplication" may negatively affect the code coverage. As our secondary criteria focus on test smells, we expect the size of the tests to increase (at least during the search process), which may promote more diversity and therefore increase coverage.

Pitfall 3 – Runtime (fewer generations)

Compared to optimizing test case length as the secondary criterion, the optimization of test smell metrics might be slower. EvoSuite has a limited search budget, e.g., the search process stops when a specified amount of time runs out. During the search, the population evolves to optimize objectives related to code coverage. The longer the optimization of test smell metrics as secondary criteria takes, the less time there is for the population to evolve and, as such, the lower the code coverage might be. This problem is unavoidable as the computation of test smell metrics takes longer (given its complexity) than the procedure to compute the test case length. We, therefore, must ensure that the optimization of test smell metrics is as fast as possible to avoid a significant impact on the effectiveness of the generated tests. Alternatively, one could opt for an approach like the one used by Grano et al. [45] and only enable the secondary criteria after a certain amount of time of the search has passed (which might reduce any negative impact on the coverage).

Approach Details

We consider each test smell as an independent secondary criterion so that anyone using EvoSuite can choose any combination of test smells they find most suitable. In particular, the primary motivation for this design decision is that we seek to find the ideal combination of test smell metrics to optimize[2]. However, we had to make two corrections in the default version of EvoSuite before performing our investigation: one of the problems prevented users from setting the secondary criteria from the command line, and the other impeded the usage of multiple secondary criteria simultaneously.

Let c_a and c_b be the chromosomes under analysis and let $S = \{s_1, ..., s_N\}$ denote the chosen secondary criteria. If both chromosomes are equally good in terms of coverage, EvoSuite uses the secondary criteria to prioritize tests with other qualities. Our version of this process works as follows:

$$compare(c_a, c_b) = \sum_{s_k \in S} s_k(c_a, c_b) \tag{4.1}$$

where $s_k(c_a, c_b) \in [0, 1]$ — therefore, all secondary criteria have the same range, so it becomes possible to determine how much better/worse c_a is than c_b when considering a set of test smells with intrinsically different metrics. Indeed, we had to make this change because, as already established, we intend to optimize metrics instead of the actual test

[2]From this point forward, although it is possible to combine any test smell with other unrelated secondary criteria, we will assume that all chosen secondary criteria correspond to test smells.

smells; as such, given that our approach does not use thresholds, we decided to normalize the smelliness of each test smell metric to balance out the different types of smells. Each test smell metric is optimized as follows:

$$s_k(c_a, c_b) = smell\_value_{c_a}(k) - smell\_value_{c_b}(k) \tag{4.2}$$

where $smell\_value_{c_a}(k)$ denotes the smelliness of the metric k for the chromosome c_a, and $smell\_value_{c_b}(k)$ denotes the smelliness of the metric k for the chromosome c_b. After iterating over S, if $compare(c_a, c_b) < 0$, then c_a is a better solution than c_b.

Note that to ensure that the secondary criteria are computed as fast as possible, given the fact that EvoSuite has a limited search budget, each test chromosome stores the results of its computed test smell metrics. Hence, unless a chromosome is modified by the search procedure, the metrics are only computed once.

4.3.3 Optimize Test Smell Metrics as Post-Processing Steps

EvoSuite applies multiple post-processing steps after the search to improve the quality of the generated tests [71, 93]. Specifically, EvoSuite [25]: (1) inlines primitive values and null references; (2) minimizes the tests; (3) adds a minimized set of assertions to each test case. The post-processing steps are independent of the test generation process, so they can optimize non-coverage metrics without interfering with the final code coverage [25, 46].

As suggested by Grano et al. [46], it should be possible to optimize test smell metrics as additional post-processing steps. In fact, the default post-processing steps applied by EvoSuite can already make the generated tests (potentially) less smelly [93]. While such an approach is resource intensive, it does not interfere with the evolutionary process. Moreover, by optimizing the proposed metrics as additional post-processing steps, we can optimize more test smell metrics than if we were to use secondary criteria (e.g., test smells related to assertions).

Initially, we considered exploring a post-processing technique similar to the one used by Daka et al. [25]:

Since automatically generated tests tend to be particularly difficult to read, Daka et al. [25] incorporated a domain-specific readability model based on human judgments into the EvoSuite tool. The researchers started by developing a readability model that uses 24 syntactic features to predict test readability. Subsequently, Daka et al. incorporated the proposed readability model into EvoSuite as an additional post-processing step so as not to negatively impact the final code coverage. The newly implemented post-processing step creates a set of alternative versions of a minimized test case and selects the candidate test case that has the best readability. The tests generated with this new approach displayed enhanced readability and high code coverage. Moreover, the researchers validated the proposed approach with human participants and observed that: (1) the tests with improved readability were usually preferred; (2) the participants answered maintenance

questions related to the enhanced tests faster and with the same accuracy. Afterward, Daka et al. [24] further investigated the proposed approach and confirmed that the readability model improves test readability without compromising coverage.

However, this approach has one detrimental issue — it is necessary to identify a viable way to generate equivalent versions of the tests that are sufficiently different to ensure a significant variation in smelliness. To better understand this problem, let us consider two ways to generate alternative tests:

Approach 1 – Generate alternative versions of the tests

We could opt for an approach like the one used by Daka et al. [25] to generate alternative versions of the minimized test cases by determining all viable replacements for each statement of each test case in the final test suite. Therefore, it would be possible to optimize the test smell metrics by selecting the least smelly candidate test cases. However, as stated by Grano et al. [46], EvoSuite generates smelly tests because the initial population is randomly generated; moreover, if the structure of the tests does not change, the tests remain smelly. Thus, replacing specific statements will not significantly impact the overall smelliness because the structure of the tests remains unchanged. For example, replacing the statements of a test case that verifies too much functionality will not reduce the smelliness. Overall, not only would this approach require a substantial amount of time/resources, but it would probably lead to negligible improvements (if any).

Approach 2 – Generate new equivalent tests

Instead of generating alternative tests by changing the test cases of the final test suite, we could generate new equivalent test suites by executing the search process with a different initial population. However, not only would this consume a lot of time, but EvoSuite does not even provide such functionality. Indeed, if we were to execute the search process all over again with a different initial population, the newly generated test suite would likely not even have the same coverage as the original test suite, so it would be better to select the one with the highest code coverage (which would defeat the entire purpose of this approach).

Problem overview

We believe that the optimization of test smell metrics as additional post-processing steps corresponds to a promising way to generate less smelly tests. However, the EvoSuite tool does not currently provide a viable way to create alternative tests with the desired characteristics. Therefore, this approach is not viable. Still, we have already implemented the test smell metrics required to perform the optimization. As such, if it becomes possible

to generate equivalent test suites with different test cases in the future, then it would be possible to use the proposed metrics to optimize test smells.

4.4 Features to Prevent Test Smells

Instead of directly optimizing test smell metrics through a fitness function (e.g., code coverage) or secondary criteria, EvoSuite could be adapted to generate tests such that the resulting outcome no longer contains certain smells. Specifically, if EvoSuite deliberately generates tests with characteristics that can be considered smelly, we could, in theory, modify the tool to not introduce such smells. For instance, although we cannot optimize smells related to tests with meaningless/unclear names, Daka et al. [26] have been able to implement a naming technique into EvoSuite to generate test cases with descriptive names, i.e., by changing the way EvoSuite creates names, the authors have eliminated the test smell "Anonymous Test".

4.5 Summary

In this chapter, we have established the set of 16 test smells that: (1) can affect the tests generated by EvoSuite; (2) we can describe with optimizable metrics. We then defined metrics for the 16 selected test smells and integrated those metrics into EvoSuite. Lastly, we investigated three approaches to optimize test smell metrics and ultimately decided to optimize test smell metrics as secondary criteria. Overall, we have established the test smell metrics we intend to optimize and the approach we intend to use to optimize said metrics; as such, we have everything we need to start implementing our solution.

Chapter 5

Empirical Study

In this chapter, we aim to investigate the following research questions:

RQ1: To what extent are the tests generated by the EvoSuite tool affected by test smells?
RQ2: What combination of test smell metrics leads to the generation of the most effective (coverage and fault detection wise) and the least smelly tests?
RQ3: Does the optimization of test smell metrics lead to the generation of significantly less smelly tests?
RQ4: Does the optimization of test smell metrics affect the fault detection effectiveness, code coverage, or size of the generated tests?

Firstly, we analyze the diffusion of the non-excluded test smells (see Section 4.1 for more details) and the distribution of the respective metrics (**RQ1**). After determining the types of bad programming practices that, in practice, affect the tests generated by EvoSuite, we identify the optimal combination of test smell metrics to optimize as secondary criteria, i.e., the one that leads not only to the lowest smelliness but also the highest code coverage and mutation score (**RQ2**). Subsequently, we investigate, on a different corpus, whether the optimal combination of test smell metrics found in **RQ2** leads to the generation of significantly less smelly tests (**RQ3**). In **RQ4**, we investigate whether the fault detection effectiveness, coverage, or size of the generated tests is affected by the optimization of test smell metrics; particularly, to address this research question, we consider and answer three sub-research questions:

RQ4.1: Does the optimization of test smell metrics affect the fault detection effectiveness of the generated tests?
RQ4.2: Does the optimization of test smell metrics affect the final code coverage of the generated tests?
RQ4.3: Does the optimization of test smell metrics affect the number and the size of the generated tests?

5.1 Experimental Subjects

In this study, we use a set of 346 non-trivial Java classes extracted from 117 open-source Java projects[1] that has been used, for example, to investigate the performance of several search-based algorithms (including EvoSuite's default, DynaMOSA) [19, 20, 72].

We decided to use this corpus of non-trivial classes as several other studies have shown that the quality/complexity of the production code can affect the presence of test smells in the test suites generated by EvoSuite [46, 69]. Indeed, given that complex classes typically imply the generation of smellier tests, experiments on this corpus should provide further insight into the presence of smells in automatically generated tests (relevant for **RQ1**). This corpus of complex classes should also allow us to more thoroughly evaluate the capabilities of the proposed approach to optimize test smell metrics (relevant for **RQ2-4**).

To manage the high computational costs associated with **RQ2**, we decided to use a random subset of 34 classes from the benchmark[2]; specifically, this is the subset originally selected by Campos et al. [19] to find the best parameters for particular genetic algorithms. In turn, we exclude these 34 classes from all other research questions, i.e., we use the remaining subset of 312 classes to answer **RQ1**, **RQ3**, and **RQ4**.

5.2 Experimental Procedure

For all research questions, we generate test suites for the selected classes using EvoSuite version 1.1.1 with the following settings: (1) DynaMOSA as the search algorithm [72]; (2) the default coverage criteria, i.e., line, branch, exception, weak mutation, output, method, method exception, and cbranch coverage [34, 71]; (3) a search budget of 180 seconds [45]. Moreover, given that EvoSuite is randomized [34], we repeated each execution 30 times (as suggested by Arcuri and Briand [6]). All experiments were executed on the Faculty of Engineering of the University of Porto (Portugal) HPC Cluster [65]. Each cluster nodes was running Scientific Linux v7.9 (x86_64) and using eight cores at 2.80 GHz with 128 GB of RAM.

5.2.1 RQ1: To what extent are the tests generated by the EvoSuite tool affected by test smells?

Previous studies may have investigated the distribution of smells in the tests generated by EvoSuite [46, 69, 74, 75], but none of them have considered the same set of 16 test smells as the one defined in Section 4.1; moreover, no other study has investigated the smelliness of the tests generated by the latest version of EvoSuite (i.e., v1.1.1). Thus, in **RQ1**, we

[1] https://github.com/jose/non-trivial-java-classes-to-study-search-based-software-testing-approaches

[2] https://github.com/jose/non-trivial-java-classes-to-study-search-based-software-testing-approaches/blob/master/data/classes-training.csv

study the diffusion of the 16 selected smells to identify the least/most prevalent test smells. To answer **RQ1**, we first implement the metrics for the selected test smells (as defined in Section 4.2). Subsequently, we study: (1) the distribution of the test smell metrics; (2) the diffusion of all types of test smells — we need to run EvoSuite on the subset of 312 classes from the benchmark and analyze the smelliness before and after post-processing is applied because test smells can be introduced/removed during the post-processing steps [93].

For each class c in the subset of 312 selected classes C, we generate 30 different test suites (R_c); for each test suite, we calculate the smelliness of each of the 16 test smell metrics. Let $T = \{t_1, ..., t_N\}$ denote the test cases in a test suite generated by EvoSuite for a class c in a run r and let $S = \{s_1, ..., s_M\}$ designate the set of test smell metrics under analysis. In each run of class c, we calculate the smelliness of a test suite T; hence, for each test case t, we obtain a list containing the smelliness of each test smell metric s.

Firstly, we investigate the distribution of the test smell metrics. Given a test suite T, we calculate the smelliness of a specific test smell metric s as follows:

$$\bar{s}(T) = \frac{1}{|T|} \sum_{t \in T} t(s) \tag{5.1}$$

where $t(s)$ denotes the smelliness of a specific test smell metric s in a test case t. Since run EvoSuite on each of the 312 considered classes (C) 30 times with different random seeds (R_c), we compute the **average smelliness** of each test smell metric $\bar{s}$ as follows:

$$\bar{s} = \frac{1}{|C|} \sum_{c \in C} \left(\frac{1}{|R_c|} \sum_{T \in R_c} \bar{s}(T) \right) \tag{5.2}$$

In turn, to calculate the overall smelliness of a test suite T, for each test case t, we need to compute the smelliness of all test smell metrics S:

$$\bar{t}(S) = \frac{1}{|S|} \sum_{s \in S} t(s) \tag{5.3}$$

where $t(s)$ denotes the smelliness of a specific test smell metric s in a test case t. Given a test suite T, we calculate the overall smelliness of all test smell metrics S as follows:

$$\bar{T}(S) = \frac{1}{|T|} \sum_{t \in T} \bar{t}(S) \tag{5.4}$$

Since we have to run EvoSuite on each of the 312 considered classes (C) 30 times with different random seeds (R_c), we compute the **average overall smelliness** $\bar{S}$ as follows:

$$\bar{S} = \frac{1}{|C|} \sum_{c \in C} \left(\frac{1}{|R_c|} \sum_{T \in R_c} \bar{T}(S) \right) \tag{5.5}$$

We decided to use the thresholds established in Section 4.2 to calculate the percentage of test cases in a given test suite affected by a specific test smell (i.e., investigate the

diffusion of test smells). To perform this analysis, we apply the thresholds to each test smell metric s of each test case t in a test suite T and calculate the percentage of test cases affected by s:

$$\text{ratio}(T, s) = 100 \times \frac{1}{|T|} \sum_{t \in T} \begin{cases} 1 & \text{if } t(s) \text{ is above or equal to threshold} \\ 0 & \text{if } t(s) \text{ is below threshold} \end{cases} \tag{5.6}$$

Given a test smell metric s, we calculate the **percentage of smelly test cases per test suite** (i.e., the ratio of smelly test cases) as follows:

$$\text{ratio}(s) = \frac{1}{|C|} \sum_{c \in C} \left(\frac{1}{|R_c|} \sum_{T \in R_c} \text{ratio}(T, s) \right) \tag{5.7}$$

where C denotes the subset of 312 classes, and R_c corresponds to the number of runs (i.e., 30). Finally, we calculate the **overall percentage of smelly test cases per test suite** for the combination of all metrics as follows:

$$\text{ratio}(S) = \frac{1}{|S|} \sum_{s \in S} \text{ratio}(s) \tag{5.8}$$

where S denotes the set of test smell metrics under analysis.

5.2.2 RQ2: What combination of test smell metrics leads to the generation of the most effective (coverage and fault detection wise) and the least smelly tests?

Concerning **RQ2**, we perform a tuning study to identify the **optimal combination of test smell metrics to optimize as secondary criteria**, i.e., the configuration that achieves not only the lowest possible smelliness but also the highest possible coverage and mutation score. To this aim, we examine the results of **RQ1** and discard the test smells that never occur or only affect a negligible portion of the final test suites generated by EvoSuite[3]; in particular, we decided to exclude the test smells that affect less than 0.50% of test cases in a post-processed test suite (Equation 5.7). Given the set of smells that affect the tests generated by EvoSuite in practice, we identify the test smells with metrics that can be optimized as secondary criteria — these are **the test smells we intend to optimize**. We optimize all possible combinations of metrics as secondary criteria (except for the one that only optimizes the "Verbose Test" metric, which is equivalent to the default secondary criterion) and compare the respective results. To manage the high computational costs associated with the comparison of metrics, we decided to use the subset of 34 classes from the benchmark (defined in Section 5.1) originally selected by Campos et al. [19] to find the best parameters for particular genetic algorithms.

[3]The "final test suites" generated by EvoSuite correspond to test suites that, supposedly, have been correctly post-processed.

To answer **RQ2**, we compare the relative: smelliness, code coverage, and mutation score — of the tests generated by multiple versions of EvoSuite (i.e., EvoSuite augmented with different combinations of test smell metrics). Note that we optimize metrics that are related to test smells rather than actual test smells; this is relevant in the sense that a smelliness value greater than zero does not necessarily imply the existence of bad coding practices. Thus, our main objective is to use these metrics to measure the smelliness of the tests and, consequently, minimize the smelliness as much as possible. In short, what we are trying to investigate is not how close we are to generating completely smell-free tests but how the performance of the different approaches compares to the other alternatives. Hence, we decided to use relative smelliness (inspired by relative coverage [8, 19]) instead of the raw values. For the same reasons, we also use relative coverage and mutation score. Specifically, given a min and a max value, we compute a relative value as follows:

$$relative\_value(value, max, min) = \frac{value - min}{max - min} \tag{5.9}$$

Let A_c denote the set of all test suites generated for a particular class c. Given a test suite T, we calculate the relative smelliness of a specific test smell metric s as follows:

$$relative\_smell(s, A_c, T) = relative\_value\left(\bar{s}(T), min(A_c(s)), max(A_c(s))\right) \tag{5.10}$$

where $\bar{s}(T)$ designates the smelliness of a test smell metric s (Equation 5.1), $min(A_c(s))$ denotes the lowest smelliness of s measured across all runs of c, and $max(A_c(s))$ denotes the highest smelliness of s measured across all runs of c. We follow a similar procedure to calculate the **relative coverage** and the **relative mutation score**. Since we run each combination of test smell metrics 30 times (random seeds (R_c)) on the 34 classes (C), we compute the **average relative smelliness** of s as follows:

$$relative\_smell(s, A) = \frac{1}{|C|} \sum_{c \in C} \left(\frac{1}{|R_c|} \sum_{T \in R_c} relative\_smell(s, A_c, T) \right) \tag{5.11}$$

where A corresponds to the set of all test suites generated by all combinations of test smell metrics across all runs of all classes. We also follow a similar approach to calculate the **average relative coverage** and the **average relative mutation score**.

Given a test suite T and the set of test suites A_c, we calculate the **relative overall smelliness** of all test smell metrics S as follows:

$$relative\_overall\_smell(\bar{T}(S), A_c) = relative\_value\left(\bar{T}(S), min(A_c(S)), max(A_c(S))\right) \tag{5.12}$$

where $\bar{T}(S)$ designates the overall smelliness of S (Equation 5.4), $min(A_c(S))$ denotes the lowest overall smelliness measured across all runs of c, and $max(A_c(S))$ denotes the highest overall smelliness measured across all runs of c. The **average relative overall smelliness** of S is computed as:

$$relative\_overall\_smell(A, S) = \frac{1}{|C|} \sum_{c \in C} \left(\frac{1}{|R_c|} \sum_{T \in R_c} relative\_overall\_smell(\bar{T}(S), A_c) \right) \tag{5.13}$$

Given that we are trying to conjugate three different metrics, the concept of "optimal combination" is merely subjective (no combination is perfect); therefore, to determine the "best" configuration, we had to establish a selection technique that considers all three aspects. Specifically, to identify the optimal combination of test smell metrics to optimize as secondary criteria, we perform pairwise comparisons of the relative: coverage, mutation score, and overall smelliness — and, for each configuration, we report the number of tournaments won/lost (i.e., comparisons in which a given combination is statistically and significantly better/worse than another combination). Essentially, each configuration goes up against all other configurations: let A and B denote any two different configurations; for each class c, we compare A against B. We utilize the Vargha-Delaney ($\hat{A}_{12}$) [6, 90] effect size to determine which of the two configurations performs better on each class c. In addition, we use the Wilcoxon-Mann-Whitney test [6, 23] with a significance level of at least 95% ($\alpha = 0.05$) to assess whether the differences between both configurations are statistically significant. We only consider tournaments in which A is statistically and significantly better/worse than B:

- Given that we aim to maximize the code coverage and mutation score:
  - If $\hat{A}_{12} > 0.50$ and p-value < 0.05, configuration B is significantly worse than configuration A, and therefore A wins the tournament.
  - If $\hat{A}_{12} < 0.50$ and p-value < 0.05, configuration B is significantly better than configuration A, and therefore B wins the tournament.
  - If $\hat{A}_{12} = 0.50$, the two configurations achieve equal performance and, as such, neither win nor lose.
- Given that we aim to minimize the smelliness:
  - If $\hat{A}_{12} > 0.50$ and p-value < 0.05, configuration B is significantly better than configuration A, and therefore B wins the tournament.
  - If $\hat{A}_{12} < 0.50$ and p-value < 0.05, configuration B is significantly worse than configuration A, and therefore A wins the tournament.
  - If $\hat{A}_{12} = 0.50$, the two configurations achieve equal performance and, as such, neither win nor lose.

For each tournament type (i.e., coverage, mutation, and overall smelliness), we calculate the **comparison score** of each configuration, i.e., the difference between the number of tournaments won/lost. We use the tournament results to rank the configurations according to code coverage, mutation score, smelliness, and the combination of coverage, mutation, and smelliness. For the three individual ranking standards, we simply consider the respective comparison scores (i.e., a higher comparison score implies a better rank). In turn, for the overall ranking standard, we combine the comparison scores as follows:

$$\begin{aligned} comparison\_score = {} & comparison\_score(coverage) + \\ & comparison\_score(mutation) + \\ & 5 \times comparison\_score(smelliness) \end{aligned} \tag{5.14}$$

where $comparison\_score(coverage|mutation|smelliness)$ correspond to the comparison scores of the coverage, mutation, and smelliness tournament types, respectively. Note that if we were to consider that the smelliness of the tests was as important as the coverage and mutation score, we would probably either choose an overall lackluster configuration that did not excel in any aspect or a combination that improves the code coverage and mutation score but does not have any significant impact on the smelliness. Therefore, given that the primary purpose of this work is to reduce the smelliness of the generated tests, we decided to assign a greater weight to the smelliness. Specifically, we decided to multiply the smelliness comparison score by five because we wanted a high enough value to ensure that the smelliness would always be more influential to the final results[4], but not to the point where it could overshadow the other metrics. We consider the following null hypothesis in **RQ2**:

> ***RQ2:*** *No combination A performs statistically and significantly better than another combination B in terms of smelliness, coverage, or mutation score.*

It is possible to reject the null hypotheses if p-value < 0.05. In particular, since we only consider the pairwise comparisons in which a combination of metrics is statistically and significantly better/worse than another combination, as long a given configuration wins/loses one or more tournaments, we can reject the null hypothesis.

5.2.3 RQ3: Does the optimization of test smell metrics lead to the generation of significantly less smelly tests?

Concerning **RQ3**, we investigate how the optimization of test smell metrics as secondary criteria affects the smelliness of the automatically generated tests. Therefore, we need to compare the smelliness of the test suites generated by the following versions of EvoSuite:

1. **Vanilla:** The default version of EvoSuite;
2. **EvoSuite (smell-free):** EvoSuite augmented with test smell metrics optimized as secondary criteria (i.e., the version of EvoSuite that optimizes the ideal combination of test smell metrics as secondary criteria).

The main objective of this research question is to verify whether the augmented version of EvoSuite improves the smelliness of the **set of test smell metrics we intend to optimize as secondary criteria** (which includes all the non-discarded test smell metrics that could have been optimized as secondary criteria, i.e., even those not included in the optimal configuration). Thus, to answer **RQ3**, we first run EVOSUITE (SMELL-FREE) on the subset of 312 classes from the benchmark and measure the smelliness of the generated

[4]Recall that we only consider the tournaments in which one configuration is statistically and significantly better/worse than another; thus, if the comparisons associated with specific metrics tend to yield significant differences and those associated with other metrics do not, then we can expect considerable differences between the respective comparison scores.

tests before and after post-processing is applied. Subsequently, given the final test suits generated by both versions of EvoSuite, we perform a pairwise test of the **relative overall smelliness** achieved by EVOSUITE (SMELL-FREE) vs. VANILLA. We use the Vargha-Delaney ($\hat{A}_{12}$) [6, 90] effect size to assess the magnitude of the improvement in smelliness that results from optimizing the ideal combination of test smell metrics as secondary criteria:

- If $\hat{A}_{12} = 0.50$, the augmented and default versions of EvoSuite are equivalent.
- If $\hat{A}_{12} > 0.50$, the augmented version of EvoSuite generates less smelly tests than the default version of the tool.
- If $\hat{A}_{12} < 0.50$, the default version of EvoSuite generates less smelly tests than the augmented version of the tool.

To investigate whether the difference in smelliness between the two versions of EvoSuite is statistically significant, we use the non-parametric Wilcoxon-Mann-Whitney test [6, 23] with a significance level of $\alpha = 0.05$ and we consider the following null hypothesis:

> ***RQ3:*** *Our approach does not reduce the smelliness of the automatically generated unit tests.*

It is possible to reject the null hypotheses if p-value < 0.05.

To complement the results of the pairwise test, we also compare the **average overall smelliness** (Equation 5.5) and the **average relative overall smelliness** (Equation 5.13) of the final test suites generated by the two versions of the EvoSuite tool. Furthermore, for each of the individual test smell metrics we intend to optimize as secondary criteria, we compare the **average smelliness** (Equation 5.2) and the **average relative smelliness** (Equation 5.11) of the tests generated by both versions of EvoSuite.

Besides the main objective of this research question, we also intend to compare the test suites generated by the two versions of EvoSuite before/after post-processing is applied to more thoroughly assess the impact of the newly proposed secondary criteria. Furthermore, since the optimization of specific smells can have indirect effects on other test smells, we ultimately decided to analyze two main categories of test smell metrics before and after post-processing optimization:

- **Directly optimized test smell metrics –** Test smell metrics included in the optimal configuration (i.e., directly optimized as secondary criteria).
- **Indirectly optimized test smell metrics –** Test smell metrics not included in the optimal configuration, which can correspond to:
  - Test smell metrics not included in the optimal configuration but that we still intend to optimize as secondary criteria;
  - Test smells discarded in **RQ2**;
  - Test smell metrics we cannot optimize as secondary criteria.

Indeed, the main objective of this work is to improve the smelliness of the test suites generated by EvoSuite (which we assume are properly post-processed); however, to fully understand the final results, we also have to consider the characteristics of the generated tests before post-processing is applied.

To study the effects of the post-processing steps and the indirectly optimized test smell metrics, we utilize a process similar to the one used in **RQ1** to investigate the smelliness of the tests generated by the default version of EvoSuite. Specifically, given the two versions of EvoSuite, we investigate:

- The distribution of the test smell metrics:
  - Average smelliness of each test smell metric (Equation 5.2);
  - Average overall smelliness (Equation 5.5).
- The diffusion of test smells:
  - Percentage of smelly test cases per test suite (Equation 5.7);
  - Overall percentage of smelly test cases per test suite (Equation 5.8).

5.2.4 RQ4: Does the optimization of test smell metrics affect the fault detection effectiveness, code coverage, or size of the generated tests?

Regarding **RQ4**, we investigate how the optimization of test smell metrics as secondary criteria affects other relevant aspects of the generated tests (besides the smelliness). In particular, we study the: mutation score, code coverage, number of test cases (i.e., size[5]), and number of statements per test suite (i.e., length) — of the set of tests generated by the vanilla and augmented versions of EvoSuite in **RQ1** and **RQ3**, respectively. Given that neither the coverage nor the mutation score can be affected by the post-processing steps, we only have to measure these metrics after post-processing is applied. Since the effects of the post-processing steps on the size and length are irrelevant to the context of this work, we also only consider those metrics after post-processing. We answer this research question by considering three separate sub-research questions:

RQ4.1: To answer **RQ4.1**, we compare the **average mutation score** and the **average relative mutation score** of the final test suites generated by both versions of EvoSuite. We decided to use this metric since mutant detection is often considered a good measure of fault detection effectiveness [34, 40, 51].

RQ4.2: To assess whether the new secondary criteria affect the coverage achieved by the generated tests, we compare the **average coverage** and the **average relative coverage**

[5]We refer to the "size" of the tests in a general sense (includes the number of test cases and statements per test suite, as well as the average number of statements per test case). However, the "size" metric (which derives from the "Size" output variable defined by EvoSuite) only corresponds to the number of test cases.

of the tests generated by both versions of EvoSuite. Specifically, we aim to analyze the default coverage criteria used by EvoSuite.

RQ4.3: To compare the general size of the final test suites generated by both versions of EvoSuite, we use: (1) the **average size** to examine the number of test cases; (2) the **average length** to inspect the total number of statements. We also consider the percentage of test cases in a test suite affected by the "Verbose Test" smell, as well as the average smelliness of the respective metric; indeed, the "Verbose Test" should provide useful insight into the average number of statements in the test cases generated by each version of the EvoSuite tool.

Given the three considered sub-research questions, we perform pairwise comparisons of the following metrics:

- **RQ4.1:** Relative mutation score;
- **RQ4.2:** Relative coverage;
- **RQ4.3 (H_0):** Size;
- **RQ4.3 (H_1):** Length.

We use the Vargha-Delaney ($\hat{A}_{12}$) [6, 90] effect size to evaluate the magnitude of the improvement in: (1) mutation score; (2) code coverage; (3) size; (4) length — that results from optimizing the ideal combination of test smell metrics as secondary criteria:

- Given that we aim to maximize the code coverage and mutation score:
  - If $\hat{A}_{12} = 0.50$, the two versions of EvoSuite are equivalent.
  - If $\hat{A}_{12} > 0.50$, the augmented version of EvoSuite is worse than the default version of the tool with regards to the investigated metric.
  - If $\hat{A}_{12} < 0.50$, the augmented version of EvoSuite is better than the default version of the tool with regards to the investigated metric.
- Given that we aim to minimize the size and length:
  - If $\hat{A}_{12} = 0.50$, the two versions of EvoSuite are equivalent.
  - If $\hat{A}_{12} > 0.50$, the augmented version of EvoSuite is better than the default version of the tool with regards to the investigated metric.
  - If $\hat{A}_{12} < 0.50$, the augmented version of EvoSuite is worse than the default version of the tool with regards to the investigated metric.

To investigate whether the differences (related to the respective metrics) between the two versions of the EvoSuite tool are statistically significant, we use the non-parametric Wilcoxon-Mann-Whitney test [6, 23] with a significance level of $\alpha = 0.05$. We conceive four null hypotheses that state that:

> ***RQ4.1:*** *Our approach reduces the final mutation score of the automatically generated unit tests.*

> ***RQ4.2:*** *Our approach reduces the final code coverage of the automatically generated unit tests.*

RQ4.3 *(H_0): Our approach increases the size of the automatically generated unit tests.*

RQ4.3 *(H_1): Our approach increases the total length of the automatically generated unit tests.*

It is possible to reject the null hypotheses if p-value < 0.05.

5.3 Threats to Validity

Based on the guidelines reported by Wohlin et al. [94], we discuss below the threats to validity to our study.

Threats to External Validity:

Associated with the generalization of our results. We conducted our investigation on a corpus composed by 346 Java classes from 117 open-source Java projects. Although our results might not generalize to other classes/projects (e.g., industrial systems), we attempt to minimize this threat by using the largest and most diverse set of classes available and that others have used (e.g., to investigate the performance of several search-based algorithms). Moreover, our results and conclusions are limited to the tests generated by one single tool, EvoSuite. Although other tools have been proposed (e.g., Randoop [66]) EvoSuite is the only that allow us to implement the novel secondary criteria.

Threats to Internal Validity:

Associated with uncontrollable internal factors that may influence our results. Given that EvoSuite is randomized, it is necessary to run repetitions and do a statistical analysis of the data. To minimize this threat, we repeat each experiment 30 times, as suggested by Arcuri and Briand [6]. However, due to time constraints and limited resources, we have only been able to run the vanilla and augmented versions of EvoSuite (which includes **RQ1**, **RQ2**, and **RQ4**) five times. Any change performed in EvoSuite and all the scripts developed to perform the statistical analysis were reviewed by all the authors and formally tested (we have created unit tests for all implemented test smell metrics).

Threats to Construct Validity:

Associated with the correspondence between theory and observation. We optimize test smell metrics inspired by the available definitions. Furthermore, when possible, we adapt test smell metric implementations and thresholds from tools with available source code.

5.4 Summary

In this chapter, we presented four research questions and gave a detailed explanation of how we intended to approach each of them:

- RQ1: Investigate the diffusion of test smells and the distribution of the respective metrics in the tests generated by the vanilla version of EvoSuite.
- RQ2: Firstly, establish the smells that, in practice, affect the tests generated by EvoSuite. Subsequently, identify the optimal combination of test smell metrics to optimize as secondary criteria.
- RQ3: Perform a pairwise test to compare EVOSUITE (SMELL-FREE) vs. VANILLA. Moreover, investigate the directly/indirectly optimized test smell metrics.
- RQ4: Perform pairwise comparisons of the following metrics: mutation score; code coverage; size — to compare EVOSUITE (SMELL-FREE) vs. VANILLA.

Chapter 6

Results

This chapter presents and discusses the results of this study and answers the research questions proposed in Section 5.

6.1 RQ1 - Identify the Test Smell Metrics to Optimize

To investigate the extent to which the tests automatically generated by the EvoSuite tool are affected by test smells, we analyze:

- The distribution of the test smell metrics;
- The percentage of smelly test cases per test suite;
- The correlation between test smell metrics.

Table 6.1 depicts the distribution of the 16 considered test smell metrics. Given the tests generated by the vanilla version of EvoSuite, we report the average: (1) smelliness of the individual metrics; (2) overall smelliness of the combination of all metrics — before/after post-processing is applied, as well as the relative improvement in smelliness that results from applying post-processing optimization. We also report the standard deviation (σ) and confidence intervals (CI) using bootstrapping at 95% significance level. To make the distribution of each metric more perceptible, Figure 6.1 presents two types of boxplots: the left ones represent the distribution of the test smell metrics before post-processing, and the right ones represent the distribution of the test smell metrics after post-processing.

Table 6.2 reports the percentage of test cases in a test suite affected by test smells (i.e., the diffusion of test smells in the generated tests). For both individual metrics and the combination of all metrics, we present the ratio of smelly test cases before and after post-processing is applied, as well as the relative improvement in the percentage of smelly test cases that results from applying the post-processing optimization. Furthermore, we display the standard deviation (σ) and confidence intervals (CI) using bootstrapping at 95% significance level. Figure 6.2 provides a visual representation of these results.

In this section, we initially discuss the properties of the tests generated by EvoSuite before post-processing is applied. Subsequently, we investigate the characteristics of the

Table 6.1: Vanilla – Distribution of the 16 considered test smell metrics. Equation 5.2 demonstrates how to calculate the average smelliness of each test smell metric; Equation 5.5 demonstrates how to calculate the average overall smelliness.

	Before post-process			After post-process			Relative
Metric	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	improvement
AssertionRoulette	0.00	0.00	[0.00, 0.00]	0.38	2.21	[0.11, 0.54]	100.00%
DuplicateAssert	0.00	0.00	[0.00, 0.00]	0.11	1.72	[-0.09, 0.21]	100.00%
EagerTest	2.15	1.07	[2.03, 2.28]	1.23	0.78	[1.14, 1.31]	-42.64%
IndirectTesting	1.44	1.08	[1.33, 1.55]	0.58	1.09	[0.43, 0.68]	-59.77%
LackOfCohesionOfMethods	0.08	0.73	[-0.02, 0.14]	0.00	0.06	[-0.00, 0.01]	-93.97%
LazyTest	0.16	0.99	[0.02, 0.25]	0.04	0.05	[0.03, 0.05]	-74.85%
LikelyIneffectiveObjectComparison	0.00	0.00	[-0.00, 0.00]	0.00	0.00	[-0.00, 0.00]	-41.57%
ObscureInlineSetup	8.54	3.43	[8.15, 8.91]	2.54	1.57	[2.36, 2.72]	-70.22%
Overreferencing	0.31	0.29	[0.28, 0.34]	0.05	0.17	[0.03, 0.07]	-82.54%
RedundantAssertion	0.00	0.00	[0.00, 0.00]	0.00	0.00	[-0.00, 0.00]	100.00%
RottenGreenTests	2.47	1.73	[2.26, 2.65]	0.03	0.15	[0.01, 0.04]	-98.83%
TestRedundancy	0.69	5.21	[0.08, 1.23]	0.00	0.00	[0.00, 0.00]	-99.85%
UnknownTest	1.00	0.00	[1.00, 1.00]	0.46	0.27	[0.43, 0.49]	-53.87%
UnrelatedAssertions	0.00	0.00	[0.00, 0.00]	0.24	0.99	[0.10, 0.32]	100.00%
UnusedInputs	2.23	1.66	[2.05, 2.43]	0.31	0.33	[0.27, 0.34]	-86.33%
VerboseTest	12.11	3.13	[11.79, 12.46]	3.70	1.71	[3.51, 3.90]	-69.48%
Smelliness	0.28	0.02	[0.27, 0.28]	0.18	0.03	[0.17, 0.18]	-35.58%

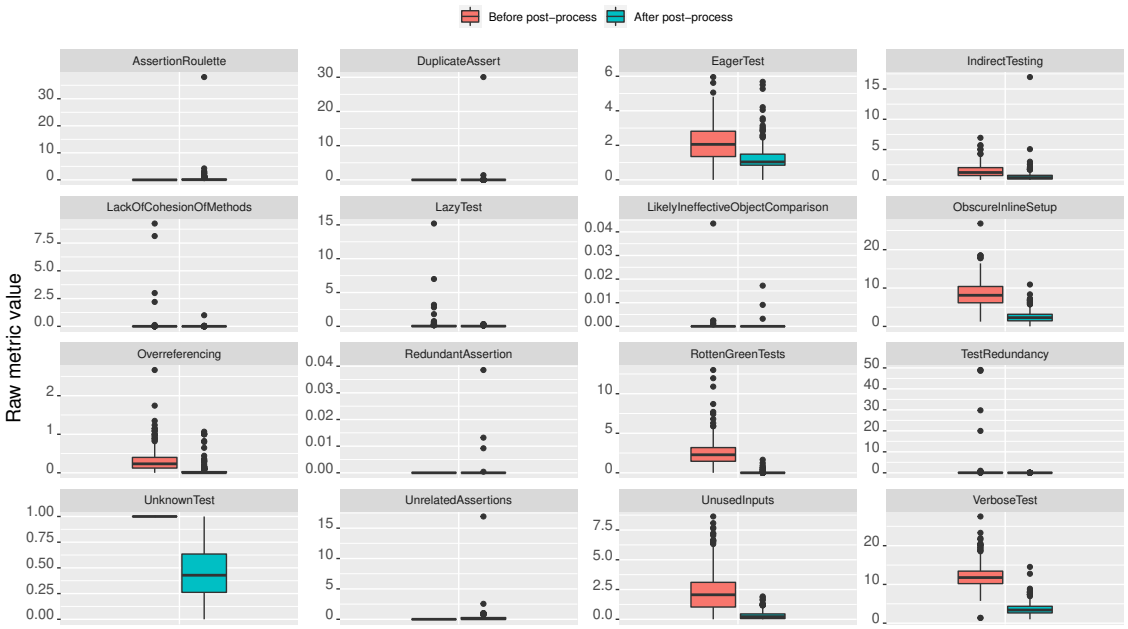

Figure 6.1: Vanilla – Distribution of metrics' raw values before/after post-processing.

generated tests after post-processing is applied and associate the observed changes with the effects of the post-processing steps.

Table 6.2: Vanilla – Diffusion of the 16 considered test smells. Equation 5.7 demonstrates how to calculate the percentage of smelly test cases per test suite; Equation 5.8 demonstrates how to calculate the overall percentage of smelly test cases per test suite.

	Before post-process			After post-process			Relative
Metric	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	improvement
AssertionRoulette	0.00%	0.00	[0.00, 0.00]	2.48%	0.09	[0.01, 0.03]	100.00%
DuplicateAssert	0.00%	0.00	[0.00, 0.00]	0.52%	0.04	[0.00, 0.01]	100.00%
EagerTest	17.72%	0.15	[0.16, 0.19]	3.73%	0.12	[0.02, 0.05]	-78.97%
IndirectTesting	46.30%	0.25	[0.44, 0.49]	35.20%	0.27	[0.32, 0.38]	-23.97%
LackOfCohesionOfMethods	0.98%	0.09	[-0.00, 0.02]	0.33%	0.06	[-0.00, 0.01]	-66.67%
LazyTest	1.37%	0.10	[0.00, 0.02]	0.00%	0.00	[0.00, 0.00]	-100.00%
LikelyIneffectiveObjectComparison	0.02%	0.00	[-0.00, 0.00]	0.01%	0.00	[-0.00, 0.00]	-41.57%
ObscureInlineSetup	29.04%	0.16	[0.27, 0.31]	1.17%	0.04	[0.01, 0.02]	-95.98%
Overreferencing	22.95%	0.17	[0.21, 0.25]	4.94%	0.15	[0.03, 0.07]	-78.46%
RedundantAssertion	0.00%	0.00	[0.00, 0.00]	0.02%	0.00	[-0.00, 0.00]	100.00%
RottenGreenTests	25.49%	0.13	[0.24, 0.27]	0.81%	0.04	[0.00, 0.01]	-96.82%
TestRedundancy	1.89%	0.13	[0.00, 0.03]	0.00%	0.00	[0.00, 0.00]	-100.00%
UnknownTest	100.00%	0.00	[1.00, 1.00]	46.13%	0.27	[0.43, 0.49]	-53.87%
UnrelatedAssertions	0.00%	0.00	[0.00, 0.00]	16.31%	0.20	[0.14, 0.18]	100.00%
UnusedInputs	69.00%	0.31	[0.65, 0.73]	25.90%	0.24	[0.23, 0.29]	-62.47%
VerboseTest	35.82%	0.13	[0.34, 0.37]	1.22%	0.05	[0.01, 0.02]	-96.60%
Average	21.91%	0.10	[0.21, 0.23]	8.67%	0.10	[0.07, 0.10]	-30.96%

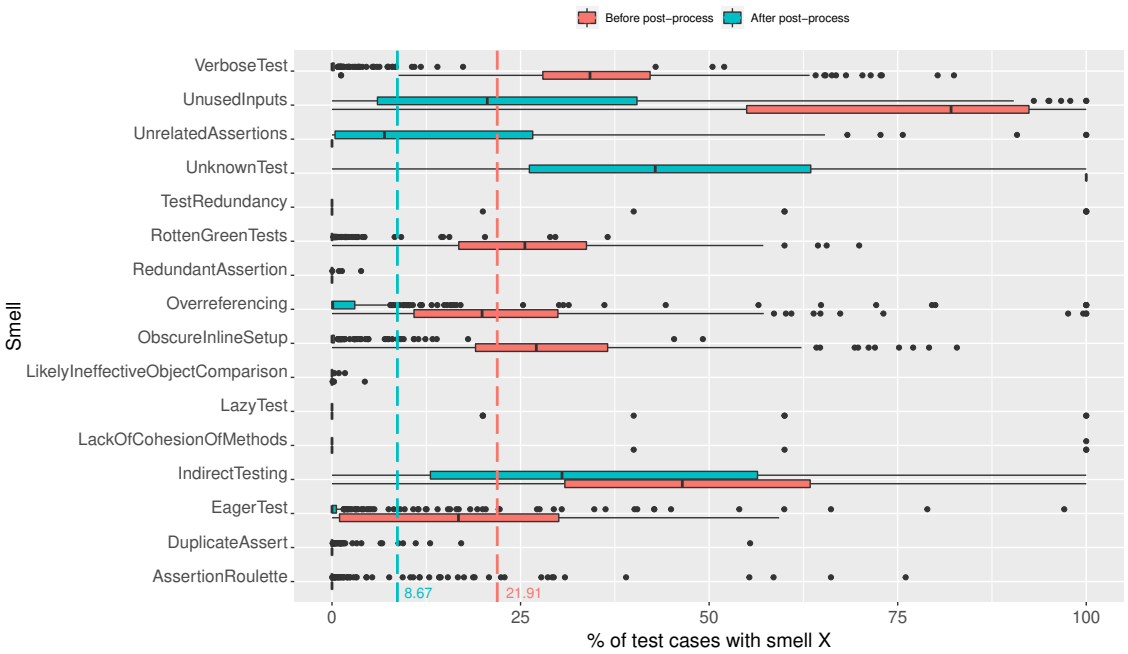

Figure 6.2: Vanilla – Distribution of the percentage of smelly test cases before and after post-processing. The vertical dashed lines represent the average smelliness before/after post-processing.

6.1.1 Before Post-Processing is Applied

First and foremost, as expected, we see that the "Assertion Roulette", "Duplicate Assert", "Redundant Assertion", and "Unrelated Assertions" smells never occur: from Table 6.1, we can observe that these metrics have an average smelliness of 0; in turn, as reported

in Table 6.2, 0% of test cases in a test suite are affected by these four smells. Indeed, it is only natural that assertion-related test smells do not arise because EvoSuite only adds assertions during the post-processing steps. The "Unknown Test" and "Unused Inputs" smells are also related to assertions, but they affect the generated tests because: (1) the "Unknown Test" smell is related to the absence of assertions in test cases; (2) the "Unused Inputs" smell is associated with statements that call methods of the class under test and that have no assertions.

The most diffused test smells unrelated to assertions: "Indirect Testing" (46.30%), "Verbose Test" (35.82%), and "Obscure In-line Setup" (29.04%). It may seem unexpected that the "Verbose Test" and, to a lesser extent, "Obscure In-line Setup" smells are not the most diffused types of test smells; however, whereas the "Indirect Testing" smell has a threshold of 1, "Verbose Test" and "Obscure In-line Setup" have thresholds of 13 and 10, respectively (see Section 4.2 for more information about the established thresholds).

The least diffused test smells unrelated to assertions: "Lazy Test" (1.37%), "Lack of Cohesion of Methods" (0.98%), and "Likely Ineffective Object-Comparison" (0.02%). Interestingly, "Likely Ineffective Object-Comparison" is the only test smell unrelated to assertions that essentially never affects the generated tests: on the one hand, as shown in Table 6.1, this smell has an average smelliness of 0; on the other hand, only 0.02% of test cases are affected by this test smell (see Table 6.2).

Before post-processing, the generated tests cannot be affected by four types of test smells. Even so, the combination of all metrics has an average overall smelliness of 0.28 (Table 6.1); moreover, 21.91% of all test cases in a test suite are smelly (Table 6.2).

6.1.2 After Post-Processing is Applied

First of all, we can observe significant general improvements that result from applying post-processing optimization: as reported in Table 6.1, the average overall smelliness decreased by 35.58% (from 0.28 to 0.18); in turn, as shown in Table 6.2, only 8.67% of the generated test cases are affected by test smells after post-processing, which indicates that the percentage of smelly test cases per test suite reduced by 30.96%. Indeed, these improvements are particularly remarkable because EvoSuite only adds assertions during the post-processing steps, which implies that the tests are susceptible to more types of test smells. Therefore, our results confirm that the post-processing steps are undoubtedly essential to ensure the generation of high-quality tests.

Regarding the test smells unrelated to assertions

As shown in Table 6.1, all test smell metrics that had an average smelliness greater than 0 before post-processing improved considerably after post-processing: in particular, the average smelliness of the "Lack of Cohesion of Methods" (reduced by 93.97%) and, more

notably, “Test Redundancy” (reduced by 99.85%) test smell metrics decreased to 0 after post-processing. Moreover, as reported in Table 6.2, the smells that affected the generated tests before post-processing also greatly improved; notably, after post-processing, 0% of test cases in a test suite are affected by the “Lazy Test” and “Test Redundancy” smells. The observed improvements may occur because EvoSuite:

- Removes redundant statements from test cases, which may improve the “Eager Test”, “Indirect Testing”, “Lazy Test”, “Likely Ineffective Object-Comparison”, “Overreferencing”, and “Verbose Test” smells.
- Removes redundant test cases, which may improve the “Indirect Testing”, “Lack Of Cohesion Of Methods”, “Lazy Test”, and “Test Redundancy” smells.
- Inlines all primitive values and null references, which may improve the “Obscure In-line Setup” and “Verbose Test” smells.
- Removes any code after the first statement that raises an exception in a test case, which may improve the “Rotten Green Tests” smell.

Surprisingly, 0.33% of test cases in a test suite are affected by the “Lack Of Cohesion Of Methods” smell after post-processing, i.e., some test cases that did not exercise the class under test were not removed by the post-processing steps. Most likely, EvoSuite did not remove such test cases because the minimization process reached its timeout.

Despite the reported improvements, we can observe that the “Indirect Testing” smell remains highly diffused (35.20%). Specifically, the “Indirect Testing” smell is by far the most diffused test smell unrelated to assertions (the second most diffused test smell, “Overreferencing”, affecting only 4.94% of test cases); moreover, this is also the second most diffused of all test smells.

Regarding the test smells related to assertions

Firstly, let us consider the assertion-related smells that affected the generated tests before post-processing, i.e., the “Unknown Test” and “Unused Inputs” test smells. As reported in Table 6.2, the “Unknown Test” (46.13%) is the most diffused of all test smells; in turn, the “Unused Inputs” (25.90%) is the third most diffused test smell. The high percentage of test cases affected by these two smells is most likely related to the existence of test cases with exceptions (which do not have assertions).

Out of the newly introduced test smells, “Unrelated Assertions” (16.31%) corresponds to the most diffused test smell; indeed, as other researchers had stated [74, 75], EvoSuite generates many test cases with assertions that check methods not declared in the class under test. The high percentage of test cases affected by the “Unrelated Assertions” smell is most likely related to the high diffusion of the “Indirect Testing” smell: test cases that check methods not declared in the class under test are also likely to have assertions that check said methods (not declared in the class under test). In turn, as reported in Table 6.1, the “Redundant Assertion” smell is the only assertion-related test smell metric with an

average smelliness of 0; moreover, only 0.02% of test cases are affected by this test smell. Additionally, the "Assertion Roulette" smell only affects 2.48% of the generated test cases because, to avoid overlap between this smell and the "Eager Test" smell, we only consider the assertions in a test case that exceed the total amount of statements that call methods of the class under test (see Section 4.2 for more details).

RQ1: On average, 8.67% of the test cases generated by the EvoSuite tool are smelly; however, if not for the post-processing steps, the percentage of smelly test cases would be much higher. The final test suites generated by EvoSuite are primarily affected by the "Unknown Test", "Indirect Testing", and "Unused Inputs" test smells.

6.2 RQ2 - Finding the Ideal Combination of Metrics

6.2.1 Test Smell Metrics to Optimize

To establish the set of test smell metrics to be optimized, we intend to exclude the smells that affect an insignificant fraction of the generated test cases, i.e., we want to discard the trivially diffused test smells. In particular, we decided to exclude the smells that affect less than 0.50% of test cases in a post-processed test suite. By analyzing the ratio of smelly test cases after post-processing is applied (Table 6.2), we can identify the test smells that we should discard: "Lack Of Cohesion Of Methods" (0.33%), "Lazy Test" (0.00%), "Likely Ineffective Object Comparison" (0.01%), "Redundant Assertion" (0.02%), and "Test Redundancy" (0.00%). The exclusion process of the test smells that do not occur in practice is represented in the 3rd step of Figure 4.1.

Test smell metrics that can be optimized as secondary criteria

Given the initial set of 16 test smells, we have identified 5 (trivially diffused) test smells that affect less than 0.50% of the test cases in a test suite. Out of the 11 remaining test smells, it is possible to optimize the following metrics as secondary criteria:

1. Eager Test;
2. Indirect Testing;
3. Obscure In-line Setup;
4. Overreferencing;
5. Rotten Green Tests;
6. Verbose Test.

The exclusion process of the test smell metrics that cannot be optimized as secondary criteria is represented in the 4th step of Figure 4.1. Moreover, the six test smell metrics we intend to optimize as secondary criteria are highlighted in gray in Table 4.1.

6.2.2 Optimal Configuration to Optimize as Secondary Criteria

Given the bad smells that affect a significant portion of the tests generated by EvoSuite, we have selected six metrics to optimize as secondary criteria: “Eager Test”; “Indirect Testing”; “Obscure In-line Setup”; “Overreferencing”; “Rotten Green Tests”; “Verbose Test”. Considering that simply optimizing all six test smell metrics as secondary criteria might not be the best solution to our problem, we decided to go the extra mile and search for the best possible metric combination to use with our approach.

For each combination of test smell metrics, Table A.1 reports the average raw/relative: code coverage, mutation score, and overall smelliness — and their respective standard deviations (σ) and confidence intervals (CI). We also report the average raw/relative smelliness of the individual test smell metrics. Across all configurations:

- The worst measured average relative code coverage is 0.65, and the best is 0.74.
- The worst measured average relative mutation score is 0.54, and the best is 0.61.
- The worst measured average relative smelliness is 0.44, and the best is 0.32.

To identify the optimal combination of test smell metrics to optimize as secondary criteria, we first perform pairwise comparisons of the relative: code coverage, mutation score, and overall smelliness. Subsequently, for each of these three types of tournaments, we calculate the **comparison score** of each configuration. Finally, we use the tournament results (i.e., the comparison scores) to rank the configurations according to code coverage, mutation score, smelliness, and the combination of all three metrics.

Table 6.3 presents the results of the pairwise tournaments. Specifically, for each of the four ranking standards, we report the top-10 best metric combinations: the configurations are ranked such that the best configuration occupies the highest position in the respective standard. Additionally, for each configuration, we report the total number of tournaments won/lost — the comparison score of each of the three tournament types corresponds to the difference between the number of tournaments won/lost.

Selecting the optimal configuration to optimize as secondary criteria

As shown in the overall ranking standard of Table 6.3, the best overall configuration is the one that optimizes the “Eager Test”, “Indirect Testing”, “Obscure In-line Setup”, and “Verbose Test” metrics; moreover, this is also the ninth best configuration in the code coverage ranking standard and the tenth best configuration in the mutation score ranking standard. Interestingly, the “ET, IT, OISS, and VT” configuration does not rank among the ten best combinations of test smell metrics in the smelliness ranking standard; still, whereas this configuration has a smelliness comparison score of 341 (because it won 433 tournaments and lost 92), the tenth best configuration in the smelliness ranking standard (“ET, IT, OISS, and RGT”) has a smelliness comparison score of 367 (won 414 tournaments and lost 47), so the difference between the two is not substantial (in fact, the

Table 6.3: Top-10 of 132,804 pairwise tournaments. The *Coverage*, *Mutation*, and *Smelliness* columns report the number of tournaments won/lost. Example: ET, RGT, and VT is the best configuration in the coverage ranking standard because it statistically won 877 tournaments and lost 46 out of 132,804.

Configuration	Coverage better on	$\hat{A}_{12}$	Coverage worse on	$\hat{A}_{12}$	Mutation better on	$\hat{A}_{12}$	Mutation worse on	$\hat{A}_{12}$	Smelliness better on	$\hat{A}_{12}$	Smelliness worse on	$\hat{A}_{12}$
ranked by Coverage												
ET, RGT, and VT	877	0.75	46	0.27	503	0.72	19	0.23	194	0.23	356	0.72
OISS, RGT, and VT	745	0.74	56	0.27	408	0.71	36	0.26	225	0.23	291	0.71
OISS, OF, RGT, and VT	712	0.73	65	0.24	390	0.71	67	0.28	220	0.23	362	0.72
IT, OISS, RGT, and VT	670	0.74	69	0.29	298	0.72	35	0.25	452	0.24	173	0.72
ET, OF, and VT	638	0.72	66	0.25	321	0.71	57	0.28	180	0.23	435	0.72
RGT	711	0.81	195	0.28	521	0.78	150	0.34	39	0.23	1025	0.83
ET, IT, and OISS	568	0.73	67	0.24	276	0.71	55	0.27	409	0.24	167	0.71
ET, OISS, OF, and VT	566	0.73	72	0.24	337	0.71	68	0.27	219	0.23	307	0.72
ET, IT, OISS, and VT	532	0.73	69	0.24	236	0.71	46	0.25	433	0.24	92	0.69
OISS, OF, and RGT	593	0.75	136	0.28	378	0.73	87	0.29	245	0.23	342	0.73
ranked by Mutation												
ET, RGT, and VT	877	0.75	46	0.27	503	0.72	19	0.23	194	0.23	356	0.72
OISS, RGT, and VT	745	0.74	56	0.27	408	0.71	36	0.26	225	0.23	291	0.71
RGT	711	0.81	195	0.28	521	0.78	150	0.34	39	0.23	1025	0.83
OISS, OF, RGT, and VT	712	0.73	65	0.24	390	0.71	67	0.28	220	0.23	362	0.72
OISS, OF, and RGT	593	0.75	136	0.28	378	0.73	87	0.29	245	0.23	342	0.73
ET, OISS, OF, and VT	566	0.73	72	0.24	337	0.71	68	0.27	219	0.23	307	0.72
ET, OF, and VT	638	0.72	66	0.25	321	0.71	57	0.28	180	0.23	435	0.72
IT, OISS, RGT, and VT	670	0.74	69	0.29	298	0.72	35	0.25	452	0.24	173	0.72
ET, IT, and OISS	568	0.73	67	0.24	276	0.71	55	0.27	409	0.24	167	0.71
ET, IT, OISS, and VT	532	0.73	69	0.24	236	0.71	46	0.25	433	0.24	92	0.69
ranked by Smelliness												
IT and OISS	159	0.72	382	0.27	58	0.68	229	0.27	542	0.25	60	0.69
IT and VT	134	0.71	296	0.27	90	0.69	194	0.28	518	0.24	38	0.68
ET, IT, and VT	163	0.71	262	0.26	111	0.69	212	0.27	504	0.24	32	0.69
IT, OISS, and VT	174	0.70	195	0.27	51	0.68	215	0.29	461	0.24	38	0.68
IT, OISS, and RGT	349	0.72	172	0.28	181	0.69	158	0.30	496	0.25	86	0.72
ET, IT, OISS, RGT, and VT	132	0.74	304	0.28	59	0.70	201	0.29	458	0.24	50	0.69
ET, IT, OISS, OF, and VT	157	0.72	269	0.27	47	0.69	253	0.28	447	0.24	56	0.69
IT, OISS, OF, and RGT	127	0.72	413	0.26	92	0.69	223	0.28	486	0.25	99	0.68
ET, IT, RGT, and VT	155	0.73	229	0.25	119	0.70	190	0.28	461	0.23	84	0.69
ET, IT, OISS, and RGT	153	0.73	242	0.27	92	0.71	185	0.28	414	0.24	47	0.68
ranked by Coverage, Mutation, and Smelliness												
ET, IT, OISS, and VT	532	0.73	69	0.24	236	0.71	46	0.25	433	0.24	92	0.69
IT, OISS, RGT, and VT	670	0.74	69	0.29	298	0.72	35	0.25	452	0.24	173	0.72
IT, OISS, and RGT	349	0.72	172	0.28	181	0.69	158	0.30	496	0.25	86	0.72
ET, IT, and VT	163	0.71	262	0.26	111	0.69	212	0.27	504	0.24	32	0.69
IT and VT	134	0.71	296	0.27	90	0.69	194	0.28	518	0.24	38	0.68
IT and OISS	159	0.72	382	0.27	58	0.68	229	0.27	542	0.25	60	0.69
ET, IT, and OISS	568	0.73	67	0.24	276	0.71	55	0.27	409	0.24	167	0.71
IT, OISS, and VT	174	0.70	195	0.27	51	0.68	215	0.29	461	0.24	38	0.68
ET, IT, RGT, and VT	155	0.73	229	0.25	119	0.70	190	0.28	461	0.23	84	0.69
ET, IT, OISS, RGT, and VT	132	0.74	304	0.28	59	0.70	201	0.29	458	0.24	50	0.69

best overall configuration won more tournaments). In turn, besides the "ET, IT, OISS, and VT" configuration, there are only two other configurations with positive smelliness comparison scores that rank among the ten best combinations of test smell metrics in the coverage and mutation ranking standards, both of which have a significantly lower smelliness comparison score than the best overall configuration: the "ET, IT, and OISS" configuration, which has a smelliness comparison score of 242; the "IT, OISS, RGT, and VT" configuration, which has a smelliness comparison score of 279.

As reported in Table A.1, the "ET, IT, OISS, and VT" configuration has an average relative code coverage of 0.71, an average relative mutation score of 0.58, and an average relative overall smelliness of 0.33; indeed, by comparing these results to the best values

measured across all configurations, we can confirm that this combination of metrics simultaneously excels in all considered aspects. Moreover, regarding the individual metrics optimized as secondary criteria, in comparison to the best configuration in the smelliness ranking standard ("IT and OISS"), the "ET, IT, OISS, and VT" configuration exhibits:

1. The same relative smelliness for the "Rotten Green Tests" metric;
2. Better relative smelliness for both the "Eager Test" (difference of 0.02) and the "Overreferencing" (difference of 0.01) metrics;
3. Worse relative smelliness for the "Indirect Testing" (difference of 0.01), "Obscure In-line Setup" (difference of 0.03) and "Verbose Test" (difference of 0.01) metrics.

Specifically, this implies that not only does the "ET, IT, OISS, and VT" configuration far outperform the best configuration in the smelliness ranking standard in terms of both code coverage and mutation score, but it is also nearly on par with it in terms of smelliness.

By analyzing the individual ranking standards, we can see some interesting patterns. First and foremost, the coverage and mutation ranking standards contain the same top-10 best metric combinations (but some configurations occupy different positions) — these results are in line with our expectations since there exists a strong positive correlation between code coverage and fault detection effectiveness (in fact, EvoSuite's default set of coverage criteria includes the weak mutation criterion). In turn, the smelliness ranking standard does not have a single configuration in common with the other two individual ranking standards; indeed, we can see that, in general, the top-10 best combinations of test smell metrics in the smelliness ranking standard have very low coverage and mutation comparison scores (in fact, the "IT, OISS, and RGT" configuration is the only one with positive code coverage and mutation comparison scores).

Overall, these results suggest that, by minimizing the smelliness of the generated tests, we also decrease the coverage and mutation score; however, the best overall configuration ("ET, IT, OISS, and VT") seems to strike a good balance between the: coverage, mutation score, and smelliness — of the generated tests. Therefore, we consider that the "ET, IT, OISS, and VT" corresponds to the optimal combination of test smell metrics to optimize as secondary criteria.

RQ2: The final test suites generated by EvoSuite are primarily affected by 11 types of test smells, six of which can be directly optimized as secondary criteria. Given the proposed selection technique, the optimal combination of test smell metrics to optimize as secondary criteria is the one that optimizes the "Eager Test", "Indirect Testing", "Obscure In-line Setup", and "Verbose Test" metrics. We selected the "ET, IT, OISS, and VT" configuration because it seems to strike a good balance between the: coverage, mutation score, and smelliness — and, as such, shows potential as a viable replacement for the default secondary criterion.

6.3 RQ3 - Smelliness Improvements

We have replaced EvoSuite's default secondary criterion with new secondary criteria that optimize test smell metrics; in particular, we decided to optimize the following metrics: "Eager Test", "Indirect Testing", "Obscure In-line Setup", and "Verbose Test". Indeed, out of all possible combinations of metrics that we could have optimized as secondary criteria, the selected combination was the one that exhibited the most potential as a viable replacement for the default secondary criterion.

To verify whether the newly proposed optimization approach induces a statistically significant reduction in the smelliness of the six test smell metrics we intend to optimize as secondary criteria, we perform a pairwise test of the relative overall smelliness achieved by EVOSUITE (SMELL-FREE) vs. VANILLA.

In this section, we first analyze the results of the pairwise test to verify if the tests generated by the augmented version of EvoSuite are less smelly than those generated by the default version of the tool; moreover, we also consider the number of classes in which one version of EvoSuite performed worse/better than the other. Subsequently, to fully understand the impact of our approach, we investigate the distribution/diffusion of the 16 considered metrics/smells before and after post-processing is applied.

Table 6.4 summarises the relevant differences between the final test suites generated by the vanilla and augmented versions of EvoSuite. Specifically, we report the average raw/relative overall smelliness of the generated tests, as well as the standard deviation (σ) and confidence intervals (CI) using bootstrapping at 95% significance level. We also present the average raw/relative smelliness of the individual test smell metrics.

Table 6.5 reports the results of the pairwise test. Given the EVOSUITE (SMELL-FREE) vs. VANILLA comparison, we present the effect size, p-value, and the relative smelliness improvement that results from optimizing test smell metrics. In turn, Table A.2 reports the number of classes in which the vanilla version of EvoSuite performs worse/better than the augmented version.

Table 6.6 depicts the distribution of the 16 considered test smell metrics in the tests generated by EVOSUITE (SMELL-FREE). We report the average: (1) smelliness of the individual metrics; (2) overall smelliness — before/after post-processing is applied, as well as the relative improvement in smelliness that results from applying post-processing optimization. In addition, we present the standard deviation (σ) and confidence intervals (CI) using bootstrapping at 95% significance level.

Table 6.7 reports the diffusion of test smells in the tests generated by the augmented version of EvoSuite. For both individual metrics and the combination of all metrics, we present the ratio of smelly test cases before and after post-processing is applied, as well as the relative improvement in the percentage of smelly test cases that results from applying the post-processing optimization. Furthermore, we display the standard deviation (σ) and confidence intervals (CI) using bootstrapping at 95% significance level.

Table 6.4: Average Raw/Relative Coverage, Mutation score, and (Overall) Smelliness per configuration.

			Coverage			Mutation									Smelliness		
Configuration	# Tests	# Length	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	EagerTest	IndirectTesting	ObscureInlineSetup	Overreferencing	RottenGreenTests	VerboseTest	$\bar{x}$	σ	CI
Raw values																	
ET, IT, OISS, and VT	56	264	0.78	0.20	[0.76, 0.81]	0.35	0.26	[0.33, 0.38]	0.46	0.19	0.61	0.03	0.00	0.74	0.34	0.06	[0.33, 0.34]
Vanilla	57	265	0.78	0.20	[0.76, 0.81]	0.35	0.26	[0.32, 0.38]	0.47	0.20	0.60	0.03	0.01	0.74	0.34	0.06	[0.33, 0.35]
Relative values																	
ET, IT, OISS, and VT	56	264	0.64	0.24	[0.61, 0.66]	0.56	0.25	[0.53, 0.59]	0.57	0.47	0.51	0.62	0.78	0.53	0.47	0.21	[0.45, 0.49]
Vanilla	57	265	0.63	0.25	[0.60, 0.66]	0.57	0.25	[0.54, 0.60]	0.58	0.56	0.50	0.61	0.78	0.52	0.52	0.20	[0.50, 0.55]

Table 6.5: Vanilla vs. Optimized. Pairwise test results for the size, length, coverage, mutation score, and smelliness metrics.

Configuration	$\bar{x}$	σ	CI	$\hat{A}_{12}$	p-value	Relative improvement
Size						
Vanilla	56.51	89.56	[45.45, 65.79]	—	—	—
ET, IT, OISS, and VT	56.13	90.55	[44.93, 65.82]	0.51	0.52	-0.66%
Length						
Vanilla	264.88	996.61	[134.95, 345.64]	—	—	—
ET, IT, OISS, and VT	264.48	1038.04	[128.62, 348.09]	0.49	0.54	-0.15%
Coverage						
Vanilla	0.63	0.25	[0.60, 0.65]	—	—	—
ET, IT, OISS, and VT	0.64	0.24	[0.61, 0.67]	0.50	0.43	1.41%
Mutation score						
Vanilla	0.57	0.25	[0.54, 0.60]	—	—	—
ET, IT, OISS, and VT	0.56	0.25	[0.53, 0.59]	0.51	0.49	-1.86%
Smelliness						
Vanilla	0.52	0.20	[0.50, 0.55]	—	—	—
ET, IT, OISS, and VT	0.47	0.21	[0.45, 0.49]	0.55	0.50	-10.13%

6.3.1 Pairwise Tournament Results

First off, as reported in Table 6.4, whereas the augmented version of EvoSuite has an average relative overall smelliness of 0.47, the vanilla version of the tool has an average relative overall smelliness of 0.52 (i.e., the average relative overall smelliness decreased by 0.05); moreover, both versions of EvoSuite have an average overall smelliness of 0.34. Regarding the results of the pairwise tournaments (Table 6.5), we can see that the overall smelliness of the tests generated by the augmented version of EvoSuite decreased by 10.13%; furthermore, the EVOSUITE (SMELL-FREE) vs. VANILLA comparison has an effect size of 0.55 and a p-value of 0.50 — given that $\hat{A}_{12} > 0.5$, we can conclude that the augmented version of EvoSuite is less smelly than the default version of the tool.

As depicted in Table A.2, EVOSUITE (SMELL-FREE) is better than VANILLA in 169 out of 307 classes (55.05% of classes); in turn, VANILLA is better in 40.07% of classes, and both versions are equivalent for 4.89% of classes. Taking these results into account:

- When EVOSUITE (SMELL-FREE) is better than VANILLA, we observe an average relative smelliness improvement of -37.04%.
- When EVOSUITE (SMELL-FREE) is better than or equal to VANILLA, we observe

an average relative smelliness improvement of -30.48%.

- When EVOSUITE (SMELL-FREE) is worse than VANILLA, we observe an average relative smelliness improvement of +70.92%.

Overall, the reported results suggest that the augmented version of EvoSuite improves the smelliness of the six test smell metrics we intended to optimize as secondary criteria. However, to better grasp the overall impact of the proposed approach to optimize test smell metrics, let us consider the smelliness of the 16 considered test smells in the tests generated by both versions of EvoSuite before/after post-processing is applied.

6.3.2 Before Post-Processing is Applied

Directly optimized test smell metrics

By comparing the distribution of the individual test smell metrics in the tests generated by the vanilla (Table 6.1) and augmented (Table 6.6) versions of EvoSuite, we can see that only two of the four metrics optimized with this approach (i.e., "Eager Test" and "Indirect Testing") exhibit a lower average smelliness in the augmented version of EvoSuite: the "Eager Test" metric improved by 0.05 and the "Indirect Testing" metric improved by 0.09.

By comparing the diffusion of the individual smells in the tests generated by the vanilla (Table 6.2) and augmented (Table 6.7) versions of EvoSuite, we observe that the percentage of test cases in a test suite affected by the "Eager Test" and "Indirect Testing" smells decreased and that the ratio of test cases affected by the "Obscure In-line Setup" and "Verbose Test" smells increased. The most significant difference between the two versions of EvoSuite is the percentage of test cases affected by the "Indirect Testing" smell, which decreased from 46.30% to 42.28% (difference of 4.02%).

Indirectly optimized test smell metrics

Regarding the "Overrefencing" and "Rotten Green Tests" metrics (i.e., the two metrics not included in the optimal configuration that we intend to optimize as secondary criteria), we can see that: (1) the distribution of both metrics increased; (2) the percentage of test cases in a test suite affected by the "Overrefencing" smell increased; (3) the percentage of test cases affected by the "Rotten Green Tests" smell decreased.

The proposed optimization approach does not jeopardize the smelliness of any of the four discarded metrics that we could have optimized as secondary criteria: on the one hand, the distribution/diffusion of the "Lack Of Cohesion Of Methods", "Lazy Test", and "Test Redundancy" metrics/smells decreased; on the other hand, the distribution/diffusion of the "Likely Ineffective Object Comparison" metric/smell remained the same.

Table 6.6: Augmented – Distribution of the 16 considered test smell metrics. Equation 5.2 demonstrates how to calculate the average smelliness of each test smell metric; Equation 5.5 demonstrates how to calculate the average overall smelliness.

	Before post-process			After post-process			Relative
Metric	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	improvement
AssertionRoulette	0.00	0.00	[0.00, 0.00]	0.40	2.06	[0.14, 0.56]	100.00%
DuplicateAssert	0.00	0.00	[0.00, 0.00]	0.12	1.66	[-0.09, 0.22]	100.00%
EagerTest	2.10	1.09	[1.98, 2.23]	1.23	0.78	[1.14, 1.32]	-41.30%
IndirectTesting	1.35	1.06	[1.24, 1.46]	0.52	0.59	[0.45, 0.58]	-61.86%
LackOfCohesionOfMethods	0.02	0.23	[-0.01, 0.04]	0.00	0.06	[-0.00, 0.01]	-79.80%
LazyTest	0.11	0.51	[0.04, 0.16]	0.04	0.04	[0.03, 0.04]	-63.85%
LikelyIneffectiveObjectComparison	0.00	0.00	[-0.00, 0.00]	0.00	0.00	[-0.00, 0.00]	-45.92%
ObscureInlineSetup	8.83	3.72	[8.38, 9.25]	2.57	1.59	[2.39, 2.76]	-70.85%
Overreferencing	0.39	0.37	[0.35, 0.43]	0.06	0.17	[0.04, 0.07]	-85.79%
RedundantAssertion	0.00	0.00	[0.00, 0.00]	0.00	0.00	[-0.00, 0.00]	100.00%
RottenGreenTests	2.50	1.86	[2.28, 2.70]	0.02	0.09	[0.01, 0.03]	-99.18%
TestRedundancy	0.66	5.11	[0.06, 1.17]	0.00	0.00	[0.00, 0.00]	-99.83%
UnknownTest	1.00	0.00	[1.00, 1.00]	0.46	0.27	[0.43, 0.49]	-53.91%
UnrelatedAssertions	0.00	0.00	[0.00, 0.00]	0.20	0.34	[0.16, 0.24]	100.00%
UnusedInputs	2.23	1.72	[2.04, 2.43]	0.31	0.35	[0.27, 0.35]	-86.12%
VerboseTest	12.40	3.43	[12.05, 12.78]	3.74	1.72	[3.55, 3.94]	-69.85%
Smelliness	0.27	0.02	[0.27, 0.28]	0.18	0.03	[0.17, 0.18]	-35.32%

Table 6.7: Augmented – Diffusion of the 16 considered test smells. Equation 5.7 demonstrates how to calculate the percentage of smelly test cases per test suite; Equation 5.8 demonstrates how to calculate the overall percentage of smelly test cases per test suite.

	Before post-process			After post-process			Relative
Metric	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	improvement
AssertionRoulette	0.00%	0.00	[0.00, 0.00]	2.73%	0.10	[0.02, 0.04]	100.00%
DuplicateAssert	0.00%	0.00	[0.00, 0.00]	0.73%	0.04	[0.00, 0.01]	100.00%
EagerTest	17.40%	0.15	[0.16, 0.19]	3.95%	0.12	[0.03, 0.05]	-77.31%
IndirectTesting	42.28%	0.24	[0.40, 0.45]	33.06%	0.27	[0.30, 0.36]	-21.81%
LackOfCohesionOfMethods	0.78%	0.08	[-0.00, 0.01]	0.33%	0.06	[-0.00, 0.01]	-58.33%
LazyTest	1.17%	0.09	[0.00, 0.02]	0.00%	0.00	[0.00, 0.00]	-100.00%
LikelyIneffectiveObjectComparison	0.02%	0.00	[-0.00, 0.00]	0.01%	0.00	[-0.00, 0.00]	-45.92%
ObscureInlineSetup	30.65%	0.17	[0.29, 0.33]	1.34%	0.05	[0.01, 0.02]	-95.64%
Overreferencing	26.80%	0.19	[0.25, 0.29]	5.22%	0.16	[0.03, 0.07]	-80.51%
RedundantAssertion	0.00%	0.00	[0.00, 0.00]	0.01%	0.00	[-0.00, 0.00]	100.00%
RottenGreenTests	25.19%	0.14	[0.24, 0.27]	0.66%	0.03	[0.00, 0.01]	-97.39%
TestRedundancy	1.76%	0.12	[0.00, 0.03]	0.00%	0.00	[0.00, 0.00]	-100.00%
UnknownTest	100.00%	0.00	[1.00, 1.00]	46.09%	0.27	[0.43, 0.49]	-53.91%
UnrelatedAssertions	0.00%	0.00	[0.00, 0.00]	16.29%	0.20	[0.14, 0.18]	100.00%
UnusedInputs	66.76%	0.31	[0.63, 0.71]	25.62%	0.24	[0.23, 0.28]	-61.62%
VerboseTest	37.13%	0.14	[0.35, 0.39]	1.40%	0.05	[0.01, 0.02]	-96.23%
Average	21.87%	0.10	[0.21, 0.23]	8.59%	0.10	[0.07, 0.10]	-30.54%

Overall Smelliness

We can see that the average overall smelliness of the combination of all metrics decreased from 0.28 to 0.27 (difference of 0.01); in turn, the ratio of smelly test cases per test suite decreased from 21.91% to 21.87% (difference of 0.04%). The diffusion of test smells and the distribution of the respective metrics improved; however, the reported

differences may not have been as substantial as one would initially expect because the size of the test cases increased. The vanilla version of EvoSuite uses test case length as the default secondary criterion, which is equivalent to optimizing the "Verbose Test" metric; therefore, considering that EVOSUITE (SMELL-FREE) optimizes other test smell metrics (besides the "Verbose Test") as secondary criteria, it seems reasonable that the vanilla version of EvoSuite (which uses a secondary criterion that exclusively focuses on producing test cases with as few statements as possible) generates tests with a lower average smelliness for the "Verbose Test" metric. As a result, the average smelliness of other test smells related to the size of test cases may also have increased. Specifically, the generation of test cases with more statements might have induced the increased average smelliness for the "Obscure Inline Setup" and "Overrefencing" metrics.

6.3.3 After Post-Processing is Applied

Directly optimized test smell metrics

The "Indirect Testing" metric is the only one of the four test smell metrics that we optimize as secondary criteria that has a lower average smelliness in the augmented version of EvoSuite: the average smelliness improved from 0.58 to 0.52 (difference of 0.06). The "Indirect Testing" is also the only smell that is less diffused in the augmented version of the tool: the percentage of test cases affected by this smell decreased from 35.20% to 33.06% (difference of 2.14%).

EVOSUITE (SMELL-FREE) has a lower average smelliness for the "Eager Test" metric before post-processing is applied; however, after post-processing, it has the same average smelliness as the default version of the tool. Interestingly, after post-processing is applied, the percentage of test cases in a test suite affected by the "Eager Test" smell is higher in the augmented version of the tool.

The average smelliness of the "Obscure Inline Setup" and "Verbose Test" metrics is still lower in the vanilla version of EvoSuite, but the difference between the two versions became smaller. Likewise, the percentage of test cases affected by these two smells is lower in the default version of the tool, but the difference is less significant.

Indirectly optimized test smell metrics

Regarding the test smells unrelated to assertions:

- The distribution of the "Overrefencing" metric is still lower in the vanilla version of the tool, but the difference became even more minute (difference of 0.01). The same applies to the ratio of smelly test cases.
- "The Rotten Green Tests" smell remains less diffused in EVOSUITE (SMELL-FREE). Furthermore, the average smelliness of the respective metric is now lower in EVOSUITE (SMELL-FREE).

- Both versions of EvoSuite have the same average smelliness for the four discarded test smell metrics unrelated to assertions. The respective test smells are also equally diffused in the two versions of the tool.

Regarding the assertion-related test smells, we can see that:

- The "Assertion Roulette" and "Duplicate Assert" are the only assertion-related test smells that are more diffused in EVOSUITE (SMELL-FREE). These are also the only two assertion-related smells with metrics that have a higher average smelliness in EVOSUITE (SMELL-FREE).
- Both versions of EvoSuite have the same average smelliness for the "Redundant Assertion", "Unknown Test", and "Unused Inputs" test smell metrics; however, the percentage of test cases in a test suite affected by the respective test smells is lower in the augmented version of EvoSuite.
- The average smelliness of the "Unrelated Assertions" test smell metric is lower in EVOSUITE (SMELL-FREE); moreover, the percentage of test cases affected by this smell is also lower in EVOSUITE (SMELL-FREE).

These results suggest that, overall, the optimization of test smell metrics as secondary criteria has positive effects on assertion-related test smells. In particular, the smelliness of the "Unrelated Assertions" test smell likely improved because we optimized the "Indirect Testing" metric as a secondary criterion; indeed, by ensuring that the generated test cases resort less often to methods not declared in the class under test, we also decrease the chances of having assertions that check methods not declared in the class under test. The reduction in the number of test cases in a test suite affected by the "Unknown Test" and "Unused Inputs" smells should be associated with the decreased distribution/diffusion of the "Rotten Green Tests" metric/smell (which was only indirectly optimized).

The smelliness of the "Assertion Roulette" test smell did not improve as originally predicted because the smelliness of the "Eager Test" smell did not improve.

6.3.4 Overall Smelliness of the Final Test Suites

According to the results of the pairwise tournaments, the final test suites generated by the augmented version of EvoSuite are less smelly than those generated by the vanilla version of the tool (relative improvement of 10.13%).

The only one of the six test smells considered in the pairwise tournament that exhibits significant improvements is the "Indirect Testing"; in turn, this improvement may have also been the cause for the significant reduction in the distribution/diffusion of the "Unrelated Assertions" metric/smell. Note that, in certain circumstances, the "Indirect Testing" smell may arise because a test case is (indirectly) checking the respective production class using methods of other classes; however, from Table 6.4, we can see that neither the final code coverage nor the mutation score decreased significantly (in fact, the final code cov-

erage increased), i.e., it was possible to achieve approximately the same results without resorting as extensively to methods not declared in the class under test. This substantial improvement is exceedingly relevant because: on the one hand, as depicted in Table 6.1, the "Indirect Testing" metric has the fourth highest average smelliness in the final test suites generated by the vanilla version of EvoSuite; on the other hand, as depicted in Table 6.2, the "Indirect Testing" is the second most diffused type of test smell in the final test suites generated by the default version of the tool. Automated test generation tools should only resort to methods not declared in the class under test when necessary; in turn, we demonstrated that it is possible to achieve similar results in EvoSuite without resorting as extensively to methods not declared in the class under test.

By optimizing test smell metrics as secondary criteria, the overall percentage of test cases in a test suite affected by test smells improved from 8.67% to 8.59%. Moreover, the augmented version of EvoSuite performed: better than the default version of the tool in 55.05% of all classes; equally to the default version of the tool in 4.89% of classes.

RQ3: The final test suites generated by EVOSUITE (SMELL-FREE) are overall less smelly than those generated by VANILLA. The selected combination of test smell metrics to optimize as secondary criteria induced a considerable improvement in the diffusion of the "Indirect Testing" and "Unrelated Assertions" smells (i.e., two of the most diffused smells in the tests generated by the vanilla version of EvoSuite).

6.4 RQ4 - Impact on the Fault Detection Effectiveness, Code Coverage, and Size

In this section, we investigate other relevant aspects of the generated tests (besides the smelliness). Specifically, we verify if the optimization of test smell metrics as secondary criteria impacts the fault detection effectiveness, coverage, or size of the generated tests. To study these metrics, we only have to consider the characteristics of the generated tests after post-processing is applied.

6.4.1 RQ4.1 - Impact on the Fault Detection Effectiveness

As reported in Table 6.4, the average relative mutation score of the tests generated by the augmented version of EvoSuite decreased from 0.57 to 0.56; moreover, both versions of EvoSuite have an average mutation score of 0.35. According to the results of the pairwise tournaments (Table 6.5), the fault detection effectiveness of the generated tests decreased by 1.86%. In turn, the EVOSUITE (SMELL-FREE) vs. VANILLA comparison has an effect size of 0.51 ($\hat{A}_{12} > 0.5$) and a p-value of 0.49. Therefore, we can conclude that the tests generated by the augmented version of EvoSuite have slightly worse fault detection effectiveness than those generated by VANILLA (but nothing substantial).

The decreased fault detection effectiveness of the final test suites generated by the augmented version of EvoSuite is most likely associated with genetic drift; indeed, we may be optimizing metrics that are inherently responsible for decreasing test diversity. This decrease is most likely related to the "Eager Test" and/or "Indirect Testing" metrics: on the one hand, the optimization of the "Eager Test" metric promotes test cases that exercise the fewest possible number of different methods of the class under test; on the other hand, the optimization of the "Indirect Testing" metric promotes test cases that avoid resorting to methods not declared in the class under test.

We expected this reduction in fault detection effectiveness to be related to a decrease in the number of generations created during the evolutionary process. However, on average, the augmented version of EvoSuite creates more generations than the vanilla version of the tool: whereas the vanilla version of EvoSuite has an average generation count of 487.72, the augmented version of EvoSuite has an average generation count of 495.33.

Overall, the optimization of test smell metrics as secondary criteria caused a decrease in the fault detection effectiveness of the generated tests. However, the difference is far too small to be considered detrimental. Nevertheless, this decrease is, to some extent, unexpected; indeed, as reported in the tuning study (RQ2), the "ET, IT, OISS, and VT" configuration was the tenth best combination of test smell metrics in terms of fault detection effectiveness. Therefore, it seems that, in general, the optimization of test smell metrics as secondary criteria has adverse effects on the fault detection effectiveness.

RQ4.1: The final test suites generated by the augmented and vanilla versions of EvoSuite achieve similar levels of fault detection effectiveness. However, the results suggest that most combinations of test smell metrics tend to have detrimental effects on the mutation score.

6.4.2 RQ4.2 - Impact on the Final Code Coverage

As reported in Table 6.4, the average relative code coverage of the tests generated by the augmented version of EvoSuite increased from 0.63 to 0.64; moreover, both versions of EvoSuite have an average code coverage of 0.78. In turn, according to the results of the pairwise tournaments (Table 6.5), the final code coverage improved by 1.41%. However, given that the EVOSUITE (SMELL-FREE) vs. VANILLA comparison has an effect size of 0.50 ($\hat{A}_{12} = 0.5$) and a p-value of 0.43, we conclude that the two versions of the tool are actually equivalent in terms of code coverage.

Considering that the test suites generated by the augmented version of EvoSuite are less smelly than those generated by the vanilla version of the tool, the fact that we also managed to maintain the final code coverage of the generated tests is a good achievement.

RQ4.2: The final test suites generated by the augmented and vanilla versions of EvoSuite are equivalent in terms of code coverage.

6.4.3 RQ4.3 - Impact on Test Size

As reported in Table 6.4, the average size of the tests generated by the augmented version of EvoSuite decreased from 57 to 56; in turn, the average length of the tests decreased from 265 to 264. According to the results of the pairwise tournaments (Table 6.5), the size of the generated tests decreased by 0.66%, and the length decreased by 0.15%. In turn, for the size, the EVOSUITE (SMELL-FREE) vs. VANILLA comparison has an effect size of 0.51; for the length, the EVOSUITE (SMELL-FREE) vs. VANILLA comparison has an effect size of 0.49. Indeed, these differences are far too small to be considered relevant. Therefore, we can conclude that the optimization of test smell metrics as secondary does not have any impact on the number of test cases per test suite or the total number of statements per test suite.

Number of statements per test case

At first glance, it may seem surprising that the difference in the average smelliness of the "Verbose Test" test smell metric is not more substantial between the two versions of EvoSuite; indeed, VANILLA only has one secondary criterion that prioritizes shorter tests (i.e., it only focuses on the number of statements per test case). However, if we take into account the type of operations performed in the post-processing steps to improve test readability, we realize that simply prioritizing tests with fewer statements is not ideal. Indeed, given that the post-processing steps already remove specific types of statements, we might as well focus on avoiding the smelly statements that are likely to persist. For instance, we know that primitive values are inlined, so, if possible, it might be better to focus on other types of statements.

RQ4.3: The size and length of final test suites generated by the augmented and vanilla versions of EvoSuite are similar.

6.5 Summary

In this chapter, we compared the tests generated by the vanilla and augmented versions of EvoSuite. Firstly, we compared the smelliness of the generated tests and observed that those generated by the augmented version of EvoSuite were overall less smelly; in particular, the "Indirect Testing' smell became significantly less diffused. Subsequently, we compared the mutation score, coverage, and size of the tests generated by both version

of EvoSuite and observed that they are on an equal footing when it comes to these three metrics. Overall, the tests generated by our version of EvoSuite are less smelly than those generated by the vanilla version of the tool and achieve similar levels of coverage and mutation score.

Chapter 7

Conclusion and Future Work

The original objective of this study was to integrate test smell metrics into EvoSuite and optimize said metrics to make the generated test code smell-free; furthermore, we were determined to achieve this goal without compromising the final code coverage or fault detection effectiveness of the generated tests.

We started this study by compiling test smells from several sources (Section 2.3) and selecting those that were relevant to the context of this work, i.e., that could both affect the tests generated by EvoSuite (Section 4.1.1) and be characterized by optimizable metrics (Section 4.1.2). After implementing the selected metrics into EvoSuite (Section 4.2), we investigated three approaches to optimize test smell metrics (Section 4.3); however, we ultimately decided to optimize test smell metrics as secondary criteria.

Research Findings

Firstly, we investigated the diffusion of test smells and the distribution of the respective metrics in the tests generated by the vanilla version of EvoSuite. We observed that, before post-processing optimization, the tests generated by EvoSuite are reasonably smelly; however, the post-processing steps significantly improve the smelliness of the generated tests. Nevertheless, the generated tests are still affected by bad programming practices. In particular, "Unknown Test", "Indirect Testing", and "Unused Inputs" correspond to the most diffused types of test smells in the final test suites generated by EvoSuite.

After determining the smells that, in practice, affect the tests generated by the vanilla version of EvoSuite, we performed a tuning study to identify the optimal combination of test smell metrics to optimize as secondary criteria, i.e., the configuration that achieves not only the lowest possible smelliness but also the highest possible coverage and mutation score. We observed that the choice of combination of test smell metrics had a significant impact on the coverage, mutation score, and smelliness of the generated tests. Although the less smelly combinations of test smell metrics also had lower code coverage and fault detection effectiveness, we managed to identify a configuration that seemed to strike a

good balance between coverage, mutation score, and smelliness. Specifically, we selected the configuration that optimizes the "Eager Test", "Indirect Testing", "Obscure In-line Setup", and "Verbose Test" metrics.

We compared the tests generated by the version of EvoSuite augmented with test smell metrics optimized as secondary criteria with those generated by the vanilla version of the tool and observed that:

- The overall smelliness of the generated tests improved considerably. In particular, we observed the most significant decrease in the diffusion of the "Indirect Testing" and "Unrelated Assertions" smells.
- The final test suites generated by the augmented and vanilla versions of EvoSuite are similar in terms of code coverage and fault detection effectiveness.
- The size and length of the generated tests are similar in both versions.

Overall, we managed to accomplish what we set out to achieve: improve the smelliness of the tests generated by the EvoSuite tool without compromising the code coverage or fault detection effectiveness of the generated tests.

Future Work

According to the proposed approach, each test smell metric corresponds to an independent secondary criterion. As such, it is possible to: (1) optimize any combination of test smell metrics; (2) incorporate new test smell metrics into EvoSuite. We conducted a tuning study to identify the "optimal" combination of test smell metrics to optimize as secondary criteria. Nevertheless, there may be other combinations of metrics that outperform the selected combination (or other metrics that we did not even consider in this study). As future work, we plan to optimize more complex test smell metrics (such as the "Test Code Duplication" smell) as secondary criteria; moreover, we want to investigate the viability of enabling the secondary criteria after a certain amount of time of the search has passed.

At first, we also intended to optimize test smell metrics as additional post-processing steps. However, we realized that, for such an approach to be viable, we would have to find a way to change the overall structure of the generated test suite without interfering with the coverage or mutation score of the generated tests. If it becomes possible to generate equivalent test suites with different test cases in the future, then it would be interesting to optimize test smell metrics during the EvoSuite's post-processing step.

Finally, considering that test smells can compromise the understandability and maintainability of the generated tests, we plan to conduct a user study with developers to quantify the true (negative) impact of test smells in day-to-day tasks.

Appendix A

Appendix

A.1 Detailed tuning results

A.2 Detailed comparison of the number of classes for which Vanilla performed worst/better than EvoSuite (smell-free)

Table A.2: Relative smelliness — Vanilla vs. EvoSuite (smell-free)

Project	# Classes	Vanilla	*other*	# Better	# Worse	# No diff.
100-jgaap	1	**0.31**	0.36	0	1	0
101-netweaver	2	**0.47**	0.49	1	1	0
102-squirrel-sql	2	**0.44**	0.51	0	2	0
103-sweethome3d	4	0.39	**0.27**	2	2	0
104-vuze	2	0.42	**0.36**	2	0	0
105-freemind	1	**0.40**	0.50	0	1	0
107-weka	3	0.54	**0.37**	3	0	0
108-liferay	2	**0.31**	0.53	0	2	0
109-pdfsam	1	**0.45**	0.46	0	1	0
10-water-simulator	2	0.66	**0.51**	2	0	0
110-firebird	2	0.68	**0.39**	2	0	0
11-imsmart	1	**0.48**	0.57	0	1	0
12-dsachat	2	**0.36**	0.67	0	2	0
13-jdbacl	2	0.45	**0.16**	2	0	0
14-omjstate	1	0.70	**0.38**	1	0	0
15-beanbin	2	**0.36**	0.48	0	2	0
17-inspirento	2	**0.37**	0.45	1	1	0
18-jsecurity	2	**0.50**	0.58	1	1	0
19-jmca	2	**0.36**	0.40	1	1	0
1-tullibee	2	**0.34**	0.38	0	2	0
21-geo-google	2	0.76	**0.56**	2	0	0
22-byuic	2	0.48	**0.37**	1	1	0
23-jwbf	2	**0.39**	0.45	1	1	0
24-saxpath	2	0.82	**0.73**	1	0	1
26-jipa	2	**0.40**	0.43	1	1	0
27-gangup	1	**0.75**	0.96	0	1	0
29-apbsmem	1	0.49	**0.38**	1	0	0
2-a4j	1	0.82	**0.62**	1	0	0
31-xisemele	1	0.42	**0.28**	1	0	0
32-httpanalyzer	2	0.50	**0.10**	2	0	0
33-javaviewcontrol	3	0.33	**0.32**	2	1	0
35-corina	1	**0.28**	0.52	0	1	0
36-schemaspy	2	0.71	**0.71**	1	1	0
37-petsoar	2	**0.37**	0.45	1	1	0

Table A.2: Relative smelliness — Vanilla vs. EvoSuite (smell-free)

Project	# Classes	Vanilla	*other*	# Better	# Worse	# No diff.
38-javabullboard	2	0.40	**0.30**	2	0	0
39-diffi	2	0.50	**0.47**	2	0	0
40-glengineer	2	0.50	**0.39**	2	0	0
41-follow	1	**0.43**	0.67	0	1	0
42-asphodel	1	1.00	1.00	0	0	1
44-summa	3	0.61	**0.56**	1	1	1
45-lotus	2	0.54	**0.38**	1	1	0
46-nutzenportfolio	2	**0.41**	0.53	0	2	0
47-dvd-homevideo	3	**0.58**	0.76	0	2	1
49-diebierse	1	**0.55**	0.87	0	1	0
4-rif	1	**0.15**	0.48	0	1	0
50-biff	1	0.97	**0.77**	1	0	0
51-jiprof	4	**0.43**	0.44	2	2	0
52-lagoon	2	**0.58**	0.69	1	1	0
54-db-everywhere	1	0.33	**0.07**	1	0	0
55-lavalamp	1	**0.45**	0.83	0	1	0
56-jhandballmoves	2	0.74	**0.54**	1	0	1
57-hft-bomberman	2	0.73	**0.42**	2	0	0
58-fps370	2	1.00	1.00	0	0	2
59-mygrid	2	**0.27**	0.45	0	2	0
5-templateit	1	**0.57**	0.72	0	1	0
60-sugar	2	**0.45**	0.57	0	2	0
61-noen	1	0.64	**0.49**	1	0	0
63-objectexplorer	2	0.61	**0.50**	1	0	1
64-jtailgui	2	**0.32**	0.41	0	2	0
65-gsftp	1	**0.68**	0.74	0	1	0
66-openjms	2	**0.47**	0.51	1	1	0
68-biblestudy	2	0.76	**0.55**	2	0	0
69-lhamacaw	2	0.42	**0.34**	1	1	0
71-ext4j	2	0.61	**0.54**	1	1	0
72-battlecry	2	**0.49**	0.77	0	1	1
73-fim1	1	0.48	**0.10**	1	0	0
74-fixsuite	2	**0.42**	0.48	1	1	0
75-openhre	2	**0.47**	0.55	0	2	0
77-io-project	1	**0.22**	0.38	0	1	0
78-caloriecount	3	0.61	**0.41**	2	1	0
79-twfbplayer	2	0.40	**0.32**	1	1	0
7-sfmis	1	**0.55**	0.57	0	1	0
80-wheelwebtool	3	0.47	**0.33**	2	1	0
81-javathena	3	**0.43**	0.45	1	2	0
82-ipcalculator	2	**0.41**	0.59	0	2	0
83-xbus	2	**0.43**	0.62	0	2	0
84-ifx-framework	1	**0.53**	0.75	0	1	0
85-shop	4	0.58	**0.52**	2	2	0
86-at-robots2-j	2	0.50	**0.45**	2	0	0
87-jaw-br	2	**0.70**	0.77	0	1	1
88-jopenchart	2	0.41	**0.37**	1	1	0
89-jiggler	4	0.54	**0.41**	3	1	0
8-gfarcegestionfa	2	0.67	**0.56**	1	1	0
90-dcparseargs	1	**0.51**	0.70	0	1	0
91-classviewer	2	0.79	**0.78**	1	0	1
92-jcvi-javacommon	2	0.53	**0.38**	2	0	0
93-quickserver	3	**0.44**	0.48	1	2	0
94-jclo	1	0.78	**0.47**	1	0	0
95-celwars2009	1	1.00	1.00	0	0	1
96-heal	2	0.37	**0.37**	1	1	0
97-feudalismgame	3	0.67	**0.43**	2	1	0
98-trans-locator	2	**0.72**	0.78	0	1	1
99-newzgrabber	2	0.51	**0.50**	1	1	0
checkstyle	6	0.61	**0.57**	3	2	1
commons-cli	1	0.60	**0.50**	1	0	0
commons-codec	1	0.71	**0.31**	1	0	0
commons-collections	3	0.56	**0.44**	3	0	0
commons-lang	14	0.58	**0.41**	12	2	0
commons-math	18	**0.49**	0.51	8	10	0
compiler	7	0.65	**0.42**	6	1	0

Table A.2: Relative smelliness — Vanilla vs. EvoSuite (smell-free)

Project	# Classes	Vanilla	*other*	# Better	# Worse	# No diff.
guava	10	0.51	**0.47**	6	4	0
hibernate	1	1.00	1.00	0	0	1
javaml	6	0.47	**0.45**	3	3	0
javex	1	**0.15**	0.40	0	1	0
jdom	5	0.53	**0.29**	4	1	0
jfree-chart	12	0.50	**0.36**	9	3	0
joda	13	0.64	**0.30**	13	0	0
jsci	3	**0.37**	0.53	0	3	0
scribe	6	0.52	**0.49**	4	2	0
tartarus	3	**0.37**	0.59	0	3	0
trove	9	0.48	**0.38**	6	3	0
twitter4j	7	0.52	**0.37**	6	1	0
wikipedia	4	**0.46**	0.47	2	2	0
xmlenc	1	**0.50**	0.70	0	1	0
Average		0.52	**0.47**	169 (55.05%)	123 (40.07%)	15 (4.89%)

Table A.1: Vanilla – Tuning results. Average Raw/Relative Coverage, Mutation score, and (Overall) Smelliness per configuration.

			Coverage			Mutation								Smelliness		
Configuration	# Tests	# Length	$\bar{x}$	σ	CI	$\bar{x}$	σ	CI	EagerTest	IndirectTesting	ObscureInlineSetup	Overreferencing	RottenGreenTests	VerboseTest	$\bar{x}$	σ	CI
Raw values																	
ET	35	158	0.70	0.24	[0.63, 0.78]	0.26	0.21	[0.20, 0.33]	0.44	0.20	0.60	0.04	0.01	0.72	0.33	0.06	[0.32, 0.36]
ET and IT	36	154	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.59	0.04	0.00	0.72	0.33	0.05	[0.31, 0.35]
ET, IT, and OISS	38	142	0.72	0.24	[0.64, 0.79]	0.28	0.21	[0.20, 0.35]	0.45	0.17	0.59	0.04	0.01	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, and OF	36	146	0.70	0.24	[0.63, 0.79]	0.26	0.20	[0.19, 0.33]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, OF, and RGT	37	147	0.71	0.23	[0.63, 0.78]	0.27	0.21	[0.20, 0.33]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, OF, RGT, and VT	36	142	0.71	0.24	[0.63, 0.78]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, OF, and VT	37	147	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.33]	0.44	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
ET, IT, OISS, and RGT	37	140	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.34]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, RGT, and VT	37	143	0.71	0.23	[0.64, 0.78]	0.26	0.21	[0.20, 0.33]	0.44	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OISS, and VT	38	134	0.71	0.24	[0.64, 0.80]	0.28	0.21	[0.20, 0.34]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
ET, IT, and OF	36	141	0.71	0.24	[0.63, 0.78]	0.26	0.21	[0.19, 0.33]	0.45	0.17	0.59	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OF, and RGT	36	154	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.59	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OF, RGT, and VT	37	146	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.44	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, OF, and VT	36	143	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.33]	0.44	0.17	0.58	0.03	0.00	0.72	0.32	0.06	[0.31, 0.35]
ET, IT, and RGT	37	155	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.59	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, RGT, and VT	37	148	0.71	0.24	[0.62, 0.79]	0.27	0.21	[0.20, 0.34]	0.44	0.17	0.58	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, IT, and VT	37	142	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.34]	0.44	0.17	0.58	0.03	0.00	0.71	0.32	0.06	[0.31, 0.34]
ET and OISS	37	143	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.34]	0.44	0.18	0.58	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, and OF	37	142	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.21, 0.34]	0.45	0.20	0.59	0.02	0.02	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, OF, and RGT	37	152	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.33]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, OF, RGT, and VT	37	146	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.34]	0.44	0.19	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, OF, and VT	37	139	0.72	0.24	[0.64, 0.80]	0.28	0.21	[0.20, 0.34]	0.44	0.19	0.59	0.03	0.01	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, and RGT	37	147	0.71	0.24	[0.63, 0.78]	0.27	0.21	[0.20, 0.33]	0.44	0.19	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, RGT, and VT	37	145	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.33]	0.44	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OISS, and VT	37	144	0.71	0.23	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.44	0.18	0.58	0.03	0.00	0.71	0.33	0.06	[0.31, 0.35]
ET and OF	35	146	0.70	0.24	[0.63, 0.78]	0.27	0.21	[0.20, 0.33]	0.44	0.21	0.60	0.02	0.01	0.73	0.34	0.06	[0.32, 0.36]
ET, OF, and RGT	37	159	0.72	0.24	[0.64, 0.80]	0.27	0.21	[0.20, 0.33]	0.45	0.20	0.60	0.03	0.01	0.73	0.34	0.06	[0.32, 0.36]
ET, OF, RGT, and VT	37	147	0.71	0.24	[0.63, 0.79]	0.26	0.21	[0.19, 0.33]	0.44	0.19	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET, OF, and VT	38	145	0.72	0.24	[0.64, 0.80]	0.28	0.21	[0.21, 0.35]	0.44	0.19	0.59	0.03	0.01	0.72	0.33	0.06	[0.31, 0.35]
ET and RGT	36	139	0.71	0.24	[0.64, 0.80]	0.27	0.20	[0.20, 0.34]	0.44	0.20	0.60	0.04	0.00	0.72	0.33	0.06	[0.32, 0.35]
ET, RGT, and VT	39	139	0.72	0.24	[0.65, 0.81]	0.28	0.21	[0.21, 0.35]	0.45	0.18	0.59	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
ET and VT	37	144	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.34]	0.44	0.18	0.58	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
IT	36	153	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.34]	0.46	0.16	0.59	0.04	0.01	0.72	0.33	0.06	[0.31, 0.35]
IT and OISS	36	143	0.70	0.24	[0.63, 0.78]	0.27	0.21	[0.19, 0.33]	0.45	0.17	0.58	0.03	0.00	0.71	0.33	0.06	[0.31, 0.35]
IT, OISS, and OF	36	155	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.47	0.17	0.58	0.01	0.03	0.72	0.33	0.05	[0.31, 0.35]
IT, OISS, OF, and RGT	36	155	0.70	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.33]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
IT, OISS, OF, RGT, and VT	37	146	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
IT, OISS, OF, and VT	37	145	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.33]	0.47	0.17	0.58	0.01	0.03	0.72	0.33	0.05	[0.31, 0.35]
IT, OISS, and RGT	37	145	0.71	0.24	[0.64, 0.80]	0.28	0.22	[0.20, 0.35]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
IT, OISS, RGT, and VT	38	136	0.72	0.24	[0.64, 0.80]	0.28	0.21	[0.20, 0.35]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
IT, OISS, and VT	37	143	0.71	0.24	[0.62, 0.79]	0.27	0.21	[0.19, 0.34]	0.45	0.17	0.58	0.04	0.00	0.72	0.33	0.06	[0.31, 0.34]
IT and OF	34	158	0.70	0.24	[0.63, 0.78]	0.27	0.22	[0.20, 0.34]	0.47	0.16	0.60	0.02	0.02	0.73	0.33	0.05	[0.32, 0.35]
IT, OF, and RGT	35	152	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.60	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
IT, OF, RGT, and VT	36	144	0.70	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.59	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
IT, OF, and VT	36	141	0.70	0.24	[0.63, 0.79]	0.26	0.21	[0.19, 0.33]	0.47	0.17	0.58	0.01	0.03	0.72	0.33	0.05	[0.31, 0.35]
IT and RGT	35	153	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.16	0.60	0.04	0.00	0.72	0.33	0.06	[0.31, 0.35]
IT, RGT, and VT	37	142	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.17	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.34]
IT and VT	37	143	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.33]	0.45	0.17	0.58	0.04	0.00	0.71	0.32	0.06	[0.31, 0.35]
OISS	36	153	0.70	0.24	[0.62, 0.79]	0.27	0.21	[0.19, 0.34]	0.45	0.18	0.58	0.04	0.00	0.71	0.33	0.06	[0.31, 0.35]
OISS and OF	37	153	0.71	0.24	[0.64, 0.80]	0.27	0.21	[0.20, 0.34]	0.47	0.19	0.58	0.01	0.03	0.72	0.33	0.06	[0.32, 0.35]
OISS, OF, and RGT	38	150	0.71	0.24	[0.63, 0.80]	0.28	0.21	[0.21, 0.35]	0.45	0.19	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
OISS, OF, RGT, and VT	38	134	0.72	0.24	[0.64, 0.80]	0.28	0.21	[0.21, 0.35]	0.45	0.18	0.59	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
OISS, OF, and VT	38	137	0.71	0.24	[0.64, 0.80]	0.27	0.21	[0.20, 0.34]	0.47	0.18	0.58	0.01	0.03	0.72	0.33	0.05	[0.31, 0.35]
OISS and RGT	37	154	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.33]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
OISS, RGT, and VT	38	132	0.72	0.24	[0.64, 0.80]	0.28	0.21	[0.21, 0.35]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
OISS and VT	37	138	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.18	0.58	0.03	0.00	0.71	0.33	0.06	[0.31, 0.35]
OF	34	168	0.71	0.24	[0.63, 0.79]	0.27	0.22	[0.19, 0.34]	0.46	0.21	0.60	0.02	0.02	0.73	0.34	0.06	[0.32, 0.36]
OF and RGT	35	163	0.71	0.24	[0.64, 0.79]	0.27	0.21	[0.20, 0.34]	0.46	0.20	0.60	0.03	0.01	0.73	0.34	0.06	[0.32, 0.36]
OF, RGT, and VT	37	148	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.20, 0.34]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
OF and VT	36	140	0.70	0.24	[0.63, 0.78]	0.27	0.21	[0.20, 0.34]	0.47	0.18	0.58	0.01	0.03	0.72	0.33	0.05	[0.31, 0.35]
RGT	37	157	0.72	0.24	[0.65, 0.81]	0.29	0.22	[0.21, 0.36]	0.46	0.20	0.60	0.04	0.01	0.73	0.34	0.06	[0.32, 0.36]
RGT and VT	37	148	0.71	0.24	[0.63, 0.79]	0.27	0.21	[0.19, 0.33]	0.45	0.18	0.58	0.03	0.00	0.72	0.33	0.06	[0.31, 0.35]
Relative values																	
ET	35	158	0.65	0.23	[0.58, 0.73]	0.54	0.29	[0.45, 0.64]	0.41	0.36	0.50	0.31	0.28	0.45	0.40	0.20	[0.33, 0.47]
ET and IT	36	154	0.66	0.25	[0.58, 0.75]	0.56	0.29	[0.46, 0.65]	0.43	0.29	0.48	0.29	0.28	0.44	0.36	0.20	[0.30, 0.43]
ET, IT, and OISS	38	142	0.71	0.24	[0.63, 0.80]	0.58	0.28	[0.49, 0.68]	0.44	0.29	0.45	0.28	0.28	0.42	0.34	0.18	[0.28, 0.40]
ET, IT, OISS, and OF	36	146	0.66	0.26	[0.58, 0.75]	0.55	0.29	[0.44, 0.65]	0.44	0.29	0.44	0.26	0.28	0.40	0.33	0.18	[0.27, 0.40]
ET, IT, OISS, OF, and RGT	37	147	0.67	0.25	[0.59, 0.76]	0.56	0.29	[0.47, 0.67]	0.43	0.30	0.44	0.27	0.28	0.41	0.33	0.18	[0.27, 0.39]
ET, IT, OISS, OF, RGT, and VT	36	142	0.67	0.26	[0.59, 0.76]	0.57	0.29	[0.47, 0.67]	0.43	0.30	0.44	0.26	0.28	0.41	0.33	0.18	[0.28, 0.40]
ET, IT, OISS, OF, and VT	37	147	0.67	0.26	[0.59, 0.76]	0.55	0.29	[0.44, 0.64]	0.43	0.30	0.44	0.26	0.28	0.40	0.33	0.18	[0.26, 0.39]
ET, IT, OISS, and RGT	37	140	0.67	0.26	[0.58, 0.76]	0.56	0.29	[0.46, 0.65]	0.43	0.30	0.44	0.27	0.28	0.40	0.33	0.18	[0.27, 0.39]
ET, IT, OISS, RGT, and VT	37	143	0.67	0.26	[0.58, 0.75]	0.55	0.29	[0.47, 0.65]	0.42	0.29	0.43	0.27	0.28	0.40	0.33	0.18	[0.26, 0.39]
ET, IT, OISS, and VT	38	134	0.71	0.25	[0.63, 0.79]	0.58	0.29	[0.49, 0.68]	0.43	0.30	0.44	0.27	0.28	0.41	0.33	0.18	[0.27, 0.39]
ET, IT, and OF	36	141	0.67	0.25	[0.59, 0.75]	0.56	0.29	[0.46, 0.65]	0.44	0.28	0.49	0.27	0.28	0.45	0.36	0.20	[0.29, 0.43]
ET, IT, OF, and RGT	36	154	0.67	0.25	[0.59, 0.75]	0.56	0.29	[0.46, 0.65]	0.44	0.29	0.49	0.27	0.28	0.45	0.37	0.19	[0.30, 0.42]
ET, IT, OF, RGT, and VT	37	146	0.67	0.25	[0.59, 0.76]	0.56	0.29	[0.46, 0.66]	0.42	0.30	0.45	0.26	0.28	0.41	0.33	0.18	[0.27, 0.40]
ET, IT, OF, and VT	36	143	0.68	0.25	[0.60, 0.77]	0.56	0.29	[0.47, 0.66]	0.42	0.29	0.44	0.27	0.27	0.40	0.33	0.18	[0.27, 0.39]
ET, IT, and RGT	37	155	0.68	0.24	[0.60, 0.76]	0.57	0.28	[0.48, 0.66]	0.43	0.29	0.49	0.30	0.28	0.44	0.37	0.20	[0.31, 0.43]
ET, IT, RGT, and VT	37	148	0.68	0.25	[0.60, 0.76]	0.56	0.29	[0.47, 0.65]	0.42	0.30	0.44	0.27	0.28	0.40	0.33	0.18	[0.27, 0.39]
ET, IT, and VT	37	142	0.67	0.26	[0.58, 0.76]	0.56	0.29	[0.46, 0.66]	0.42	0.29	0.43	0.28	0.28	0.39	0.32	0.18	[0.26, 0.39]
ET and OISS	37	143	0.68	0.25	[0.60, 0.76]	0.56	0.29	[0.47, 0.65]	0.42	0.32	0.44	0.28	0.28	0.40	0.34	0.18	[0.27, 0.39]
ET, OISS, and OF	37	142	0.69	0.25	[0.61, 0.77]	0.57	0.29	[0.49, 0.66]	0.44	0.36	0.46	0.23	0.31	0.44	0.37	0.17	[0.31, 0.43]
ET, OISS, OF, and RGT	37	152	0.67	0.25	[0.59, 0.75]	0.57	0.29	[0.47, 0.66]	0.44	0.32	0.45	0.26	0.28	0.42	0.35	0.18	[0.29, 0.41]
ET, OISS, OF, RGT, and VT	37	146	0.67	0.26	[0.60, 0.76]	0.56	0.29	[0.46, 0.65]	0.43	0.33	0.45	0.26	0.28	0.41	0.35	0.17	[0.29, 0.41]
ET, OISS, OF, and VT	37	139	0.71	0.25	[0.63, 0.80]	0.59	0.28	[0.49, 0.69]	0.44	0.33	0.46	0.25	0.29	0.43	0.35	0.18	[0.30, 0.42]
ET, OISS, and RGT	37	147	0.67	0.25	[0.59, 0.76]	0.55	0.29	[0.46, 0.64]	0.43	0.33	0.44	0.27	0.28	0.41	0.35	0.18	[0.29, 0.41]
ET, OISS, RGT, and VT	37	145	0.67	0.25	[0.59, 0.76]	0.56	0.29	[0.46, 0.65]	0.41	0.32	0.44	0.27	0.28	0.40	0.33	0.17	[0.28, 0.40]
ET, OISS, and VT	37	144	0.67	0.26	[0.59, 0.76]	0.56	0.29	[0.46, 0.66]	0.42	0.31	0.43	0.27	0.28	0.39	0.33	0.18	[0.27, 0.39]
ET and OF	35	146	0.67	0.23	[0.59, 0.75]	0.56	0.28	[0.46, 0.65]	0.42	0.37	0.52	0.25	0.30	0.48	0.41	0.19	[0.35, 0.48]
ET, OF, and RGT	37	159	0.71	0.23	[0.63, 0.80]	0.58	0.28	[0.48, 0.67]	0.44	0.37	0.53	0.27	0.28	0.49	0.42	0.20	[0.35, 0.48]
ET, OF, RGT, and VT	37	147	0.67	0.26	[0.58, 0.76]	0.56	0.29	[0.46, 0.65]	0.42	0.33	0.45	0.26	0.28	0.41	0.35	0.18	[0.29, 0.41]
ET, OF, and VT	38	145	0.72	0.24	[0.64, 0.80]	0.58	0.28	[0.48, 0.67]	0.43	0.34	0.47	0.25	0.30	0.44	0.36	0.17	[0.30, 0.42]
ET and RGT	36	139	0.69	0.23	[0.61, 0.77]	0.57	0.27	[0.48, 0.66]	0.43	0.35	0.51	0.29	0.28	0.47	0.40	0.20	[0.34, 0.48]
ET, RGT, and VT	39	139	0.73	0.25	[0.66, 0.82]	0.61	0.28	[0.51, 0.69]	0.44	0.32	0.47	0.28	0.28	0.43	0.36	0.19	[0.29, 0.42]
ET and VT	37	144	0.69	0.25	[0.61, 0.77]	0.56	0.29	[0.47, 0.67]	0.42	0.32	0.45	0.28	0.28	0.41	0.34	0.18	[0.28, 0.41]
IT	36	153	0.69	0.24	[0.61, 0.78]	0.58	0.29	[0.48, 0.68]	0.47	0.27	0.49	0.31	0.29	0.47	0.38	0.21	[0.30, 0.45]
IT and OISS	36	143	0.65	0.27	[0.56, 0.74]	0.55	0.29	[0.46, 0.65]	0.45	0.29	0.41	0.28	0.28	0.40	0.32	0.17	[0.27, 0.38]
IT, OISS, and OF	36	155	0.68	0.26	[0.59, 0.77]	0.56	0.29	[0.46, 0.66]	0.52	0.29	0.43	0.19	0.36	0.45	0.36	0.20	[0.29, 0.43]
IT, OISS, OF, and RGT	36	155	0.65	0.26	[0.57, 0.74]	0.55	0.29	[0.47, 0.65]	0.45	0.29	0.43	0.26	0.28	0.41	0.33	0.17	[0.27, 0.38]
IT, OISS, OF, RGT, and VT	37	146	0.68	0.25	[0.60, 0.77]	0.56	0.30	[0.45, 0.65]	0.43	0.29	0.45	0.27	0.27	0.42	0.33	0.18	[0.27, 0.39]
IT, OISS, OF, and VT	37	145	0.67	0.26	[0.58, 0.76]	0.55	0.29	[0.45, 0.64]	0.51	0.29	0.43	0.19	0.35	0.44	0.36	0.20	[0.29, 0.42]
IT, OISS, and RGT	37	145	0.69	0.25	[0.60, 0.78]	0.57	0.28	[0.48, 0.66]	0.45	0.29	0.42	0.27	0.28	0.41	0.33	0.18	[0.27, 0.39]
IT, OISS, RGT, and VT	38	136	0.71	0.25	[0.63, 0.80]	0.59	0.28	[0.49, 0.69]	0.44	0.29	0.44	0.28	0.28	0.42	0.34	0.19	[0.27, 0.40]
IT, OISS, and VT	37	143	0.68	0.25	[0.60, 0.77]	0.56	0.28	[0.46, 0.65]	0.44	0.29	0.43	0.28	0.28	0.40	0.33	0.18	[0.27, 0.38]
IT and OF	34	158	0.65	0.24	[0.57, 0.73]	0.56	0.29	[0.47, 0.65]	0.51	0.28	0.50	0.23	0.32	0.49	0.39	0.19	[0.33, 0.45]
IT, OF, and RGT	35	152	0.66	0.23	[0.58, 0.74]	0.57	0.29	[0.47, 0.66]	0.47	0.28	0.49	0.27	0.28	0.47	0.37	0.20	[0.31, 0.44]
IT, OF, RGT, and VT	36	144	0.65	0.26	[0.57, 0.74]	0.55	0.29	[0.46, 0.64]	0.44	0.29	0.45	0.26	0.28	0.42	0.34	0.18	[0.28, 0.40]
IT, OF, and VT	36	141	0.67	0.25	[0.59, 0.75]	0.55	0.29	[0.45, 0.66]	0.50	0.29	0.44	0.19	0.35	0.44	0.36	0.21	[0.29, 0.42]
IT and RGT	35	153	0.67	0.24	[0.59, 0.76]	0.57	0.28	[0.48, 0.67]	0.47	0.28	0.49	0.30	0.28	0.47	0.38	0.20	[0.31, 0.45]
IT, RGT, and VT	37	142	0.68	0.26	[0.59, 0.77]	0.56	0.29	[0.46, 0.66]	0.43	0.29	0.44	0.27	0.28	0.41	0.33	0.19	[0.27, 0.40]
IT and VT	37	143	0.67	0.26	[0.58, 0.76]	0.56	0.29	[0.47, 0.65]	0.43	0.29	0.42	0.28	0.28	0.39	0.32	0.18	[0.26, 0.38]
OISS	36	153	0.65	0.27	[0.56, 0.73]	0.55	0.29	[0.46, 0.66]	0.45	0.32	0.40	0.28	0.28	0.39	0.33	0.17	[0.27, 0.39]
OISS and OF	37	153	0.69	0.25	[0.61, 0.78]	0.57	0.29	[0.48, 0.67]	0.52	0.33	0.42	0.19	0.36	0.46	0.38	0.19	[0.31, 0.44]
OISS, OF, and RGT	38	150	0.70	0.25	[0.62, 0.78]	0.59	0.28	[0.49, 0.68]	0.46	0.32	0.44	0.26	0.28	0.43	0.35	0.18	[0.29, 0.41]
OISS, OF, RGT, and VT	38	134	0.72	0.24	[0.64, 0.81]	0.59	0.28	[0.50, 0.69]	0.45	0.32	0.46	0.27	0.28	0.43	0.35	0.18	[0.30, 0.41]
OISS, OF, and VT	38	137	0.70	0.24	[0.63, 0.79]	0.57	0.28	[0.48, 0.67]	0.51	0.32	0.43	0.18	0.36	0.44	0.37	0.19	[0.30, 0.43]
OISS and RGT	37	154	0.68	0.25	[0.61, 0.76]	0.57	0.28	[0.48, 0.66]	0.45	0.32	0.42	0.27	0.28	0.41	0.34	0.18	[0.28, 0.39]
OISS, RGT, and VT	38	132	0.72	0.24	[0.64, 0.80]	0.60	0.28	[0.51, 0.69]	0.45	0.32	0.45	0.27	0.28	0.42	0.35	0.18	[0.29, 0.41]
OISS and VT	37	138	0.68	0.25	[0.59, 0.77]	0.56	0.29	[0.48, 0.67]	0.43	0.31	0.41	0.27	0.28	0.39	0.32	0.18	[0.27, 0.38]
OF	34	168	0.68	0.22	[0.62, 0.76]	0.56	0.29	[0.47, 0.66]	0.49	0.38	0.53	0.23	0.34	0.51	0.44	0.20	[0.38, 0.51]
OF and RGT	35	163	0.69	0.24	[0.61, 0.76]	0.56	0.27	[0.47, 0.65]	0.49	0.37	0.53	0.27	0.29	0.51	0.44	0.21	[0.37, 0.51]
OF, RGT, and VT	37	148	0.67	0.26	[0.59, 0.75]	0.56	0.29	[0.45, 0.65]	0.44	0.32	0.45	0.26	0.28	0.41	0.34	0.18	[0.28, 0.41]
OF and VT	36	140	0.67	0.25	[0.59, 0.75]	0.55	0.28	[0.46, 0.65]	0.49	0.32	0.44	0.19	0.36	0.44	0.37	0.19	[0.30, 0.43]
RGT	37	157	0.74	0.23	[0.66, 0.81]	0.60	0.28	[0.51, 0.70]	0.50	0.37	0.53	0.31	0.28	0.51	0.44	0.22	[0.37, 0.52]
RGT and VT	37	148	0.68	0.25	[0.60, 0.76]	0.56	0.29	[0.47, 0.66]	0.43	0.31	0.44	0.27	0.28	0.41	0.34	0.18	[0.28, 0.39]

Bibliography

[1] Nitin Agarwal and Urvashi Rathod. Defining 'success' for software projects: An exploratory revelation. *International journal of project management*, 24(4):358–370, 2006.

[2] Wajdi Aljedaani, Anthony Peruma, Ahmed Aljohani, Mazen Alotaibi, Mohamed Wiem Mkaouer, Ali Ouni, Christian D Newman, Abdullatif Ghallab, and Stephanie Ludi. Test smell detection tools: A systematic mapping study. *Evaluation and Assessment in Software Engineering*, pages 170–180, 2021.

[3] M Moein Almasi, Hadi Hemmati, Gordon Fraser, Andrea Arcuri, and Janis Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, 2017.

[4] Saswat Anand, Edmund K Burke, Tsong Yueh Chen, John Clark, Myra B Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, Phil McMinn, Antonia Bertolino, et al. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 86(8):1978–2001, 2013.

[5] Vincent Aranega, Julien Delplanque, Matias Martinez, Andrew P Black, Stéphane Ducasse, Anne Etien, Christopher Fuhrman, and Guillermo Polito. Rotten green tests in java, pharo and python. *Empirical Software Engineering*, 26(6):1–41, 2021.

[6] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[7] Andrea Arcuri, José Campos, and Gordon Fraser. Unit test generation during software development: Evosuite plugins for maven, intellij and jenkins. In *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*, pages 401–408. IEEE Computer Society, 2016.

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

[9] Thanassis Avgerinos, Alexandre Rebert, Sang Kil Cha, and David Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1083–1094, 2014.

[10] Paris Avgeriou, Philippe Kruchten, Ipek Ozkaya, and Carolyn Seaman. Managing technical debt in software engineering (dagstuhl seminar 16162). In *Dagstuhl Reports*, volume 6. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[11] Umberto Azadi, Francesca Arcelli Fontana, and Davide Taibi. Architectural smells detected by tools: a catalogue proposal. In *2019 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pages 88–97. IEEE, 2019.

[12] Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, and Benjamin Zeiss. Trex-the refactoring and metrics tool for ttcn-3 test specifications. In *Testing: Academic & Industrial Conference-Practice And Research Techniques (TAIC PART'06)*, pages 90–94. IEEE, 2006.

[13] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.

[14] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 56–65. IEEE, 2012.

[15] Richard Berntsson-Svensson and Aybüke Aurum. Successful software project and products: An empirical investigation. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, pages 144–153, 2006.

[16] Manuel Breugelmans and Bart Van Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*. Citeseer, 2008.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, 2013.

[18] Denivan Campos, Larissa Rocha, and Ivan Machado. Developers perception on the severity of test smells: an empirical study. *arXiv preprint arXiv:2107.13902*, 2021.

[19] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, 2018.

[20] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In *International Symposium on Search Based Software Engineering*, pages 33–48. Springer, 2017.

[21] José Campos, Annibale Panichella, and Gordon Fraser. Evosuite at the sbst 2019 tool competition. In *2019 IEEE/ACM 12th International Workshop on Search-Based Software Testing (SBST)*, pages 29–32. IEEE, 2019.

[22] David M Cohen, Siddhartha R Dalal, Ajay Kajla, and Gardner C Patton. The automatic efficient test generator (aetg) system. In *Proceedings of 1994 IEEE International Symposium on Software Reliability Engineering*, pages 303–309. IEEE, 1994.

[23] William Jay Conover. *Practical nonparametric statistics*, volume 350. john wiley & sons, 1999.

[24] Ermira Daka, José Campos, Jonathan Dorn, Gordon Fraser, and Westley Weimer. Generating readable unit tests for guava. In *International Symposium on Search Based Software Engineering*, pages 235–241. Springer, 2015.

[25] Ermira Daka, José Campos, Gordon Fraser, Jonathan Dorn, and Westley Weimer. Modeling readability to improve unit tests. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 107–118, 2015.

[26] Ermira Daka, José Miguel Rojas, and Gordon Fraser. Generating unit tests with descriptive names or: Would you name your children thing1 and thing2? In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 57–67, 2017.

[27] Julien Delplanque, Stéphane Ducasse, Guillermo Polito, Andrew P Black, and Anne Etien. Rotten green tests. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 500–511. IEEE, 2019.

[28] Jon Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28. Citeseer, 1999.

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

[30] Francesca Arcelli Fontana, Paris Avgeriou, Ilaria Pigazzini, and Riccardo Roveda. A study on architectural smells prediction. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 333–337. IEEE, 2019.

[31] Francesca Arcelli Fontana, Pietro Braione, and Marco Zanoni. Automatic detection of bad smells in code: An experimental assessment. *J. Object Technol.*, 11(2):5–1, 2012.

[32] Francesca Arcelli Fontana, Marco Mangiacavalli, Domenico Pochiero, and Marco Zanoni. On experimenting refactoring tools to remove code smells. In *Scientific Workshop Proceedings of the XP2015*, pages 1–8, 2015.

[33] Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[34] Gordon Fraser. A tutorial on using and extending the evosuite search-based test generator. In *International Symposium on Search Based Software Engineering*, pages 106–130. Springer, 2018.

[35] Gordon Fraser and Andrea Arcuri. Evosuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 416–419, 2011.

[36] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2012.

[37] Gordon Fraser and Andrea Arcuri. Evosuite: On the challenges of test case generation in the real world. In *2013 IEEE sixth international conference on software testing, verification and validation*, pages 362–369. IEEE, 2013.

[38] Gordon Fraser and Andrea Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2), dec 2014.

[39] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20(3):611–639, 2015.

[40] Gordon Fraser and Andrea Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2015.

[41] Gordon Fraser and José Miguel Rojas. Software testing. In *Handbook of Software Engineering*, pages 123–192. Springer, 2019.

[42] Gordon Fraser, José Miguel Rojas, and Andrea Arcuri. Evosuite at the sbst 2018 tool competition. In *Proceedings of the 11th International Workshop on Search-Based Software Testing*, SBST '18, page 34–37, New York, NY, USA, 2018. Association for Computing Machinery.

[43] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.

[44] Gregory Gay. Generating effective test suites by combining coverage criteria. In *International Symposium on Search Based Software Engineering*, pages 65–82. Springer, 2017.

[45] Giovanni Grano, Christoph Laaber, Annibale Panichella, and Sebastiano Panichella. Testing with fewer resources: An adaptive approach to performance-aware test case generation. *IEEE Transactions on Software Engineering*, 2019.

[46] Giovanni Grano, Fabio Palomba, Dario Di Nucci, Andrea De Lucia, and Harald C Gall. Scented since the beginning: On the diffuseness of test smells in automatically generated test code. *Journal of Systems and Software*, 156:312–327, 2019.

[47] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 322–331. IEEE, 2013.

[48] Alex Gyori, August Shi, Farah Hariri, and Darko Marinov. Reliable testing: Detecting state-polluting tests to prevent test dependency. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 223–233, 2015.

[49] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[50] Chen Huo and James Clause. Improving oracle quality by detecting brittle assertions and unused inputs in tests. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 621–631, 2014.

[51] René Just, Darioush Jalali, Laura Inozemtseva, Michael D Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665, 2014.

[52] Manju Khari and Prabhat Kumar. An extensive evaluation of search-based software testing: a review. *Soft Computing*, 23(6):1933–1946, 2019.

[53] Foutse Khomh, Massimiliano Di Penta, and Yann-Gael Gueheneuc. An exploratory study of the impact of code smells on software change-proneness. In *2009 16th Working Conference on Reverse Engineering*, pages 75–84. IEEE, 2009.

[54] Fitsum M Kifetew, Annibale Panichella, Andrea De Lucia, Rocco Oliveto, and Paolo Tonella. Orthogonal exploration of the search space in evolutionary test case generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 257–267, 2013.

[55] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[56] Scott Kirkpatrick, C Daniel Gelatt, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[57] Negar Koochakzadeh and Vahid Garousi. Tecrevis: a tool for test coverage and test redundancy visualization. In *International Academic and Industrial Conference on Practice and Research Techniques*, pages 129–136. Springer, 2010.

[58] Guilherme Lacerda, Fabio Petrillo, Marcelo Pimenta, and Yann Gaël Guéhéneuc. Code smells and refactoring: A tertiary systematic review of challenges and observations. *Journal of Systems and Software*, 167:110610, 2020.

[59] Kiran Lakhotia, Mark Harman, and Phil 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, 2007.

[60] Phil McMinn. Search-based software test data generation: a survey. *Software testing, Verification and reliability*, 14(2):105–156, 2004.

[61] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[62] José Carlos Medeiros de Campos. *Search-based Unit Test Generation for Evolving Software*. PhD thesis, University of Sheffield, 2017.

[63] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[64] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[65] Faculty of Engineering of the University of Porto (FEUP). HPC Cluster.

[66] Carlos Pacheco and Michael 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*, pages 815–816, 2007.

[67] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. A large-scale empirical study on the lifecycle of code smell co-occurrences. *Information and Software Technology*, 99:1–10, 2018.

[68] Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Fausto Fasano, Rocco Oliveto, and Andrea De Lucia. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering*, 23(3):1188–1221, 2018.

[69] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. On the diffusion of test smells in automatically generated test code: An empirical study. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 5–14. IEEE, 2016.

[70] Fabio Palomba, Annibale Panichella, Andy Zaidman, Rocco Oliveto, and Andrea De Lucia. Automatic test case generation: What if test code quality matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 130–141, 2016.

[71] Annibale Panichella, José Campos, and Gordon Fraser. Evosuite at the sbst 2020 tool competition. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, pages 549–552, 2020.

[72] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, 2017.

[73] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Incremental control dependency frontier exploration for many-criteria test case generation. In *International Symposium on Search Based Software Engineering*, pages 309–324. Springer, 2018.

[74] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Revisiting test smells in automatically generated tests: Limitations, pitfalls, and opportunities. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 523–533, 2020.

[75] Annibale Panichella, Sebastiano Panichella, Gordon Fraser, Anand Ashok Sawant, and Vincent J. Hellendoorn. Test smells 20 years later: Detectability, validity, and reliability, 2022.

[76] Anthony Peruma, Khalid Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. Tsdetect: An open source test smells detection tool. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1650–1654, 2020.

[77] Anthony Peruma, Khalid Saeed Almalki, Christian D Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. On the distribution of test smells in open source android applications: An exploratory study. 2019.

[78] Anthony Peruma, Mohamed Wiem Mkaouer, Khalid Almalki, Christian D. Newman, Ali Ouni, and Fabio Palomba. Software unit test smells. `https://testsmells.org/`. Accessed: 2021-12-25.

[79] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *J. Object Technol.*, 6(9):231–251, 2007.

[80] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. Combining multiple coverage criteria in search-based unit test generation. In *International Symposium on Search Based Software Engineering*, pages 93–108. Springer, 2015.

[81] Darius Sas, Ilaria Pigazzini, Paris Avgeriou, and Francesca Arcelli Fontana. The perception of architectural smells in industrial practice. *arXiv preprint arXiv:2110.06750*, 2021.

[82] Sina Shamshiri. *Automated Unit Testing of Evolving Software*. PhD thesis, University of Sheffield, 2016.

[83] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. Random or genetic algorithm search for object-oriented test suite generation? In *Proceedings of the 2015 annual conference on genetic and evolutionary computation*, pages 1367–1374, 2015.

[84] Sina Shamshiri, José Miguel Rojas, Juan Pablo Galeotti, Neil Walkinshaw, and Gordon Fraser. How do automatically generated unit tests influence software maintenance? In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 250–261. IEEE, 2018.

[85] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. On the relation of test smells to software code quality. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 1–12. IEEE, 2018.

[86] Davide Spadini, Martin Schvarcbacher, Ana-Maria Oprescu, Magiel Bruntink, and Alberto Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, pages 311–321, 2020.

[87] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15, 2016.

[88] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*, pages 92–95. Citeseer, 2001.

[89] Eva Van Emden and Leon Moonen. Java quality assurance by detecting code smells. In *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.*, pages 97–106. IEEE, 2002.

[90] András Vargha and Harold D Delaney. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[91] Tássio Virgínio, Luana Martins, Larissa Rocha, Railana Santana, Adriana Cruz, Heitor Costa, and Ivan Machado. Jnose: Java test smell detector. In *Proceedings of the 34th Brazilian Symposium on Software Engineering*, pages 564–569, 2020.

[92] Tássio Virgínio, Railana Santana, Luana Almeida Martins, Larissa Rocha Soares, Heitor Costa, and Ivan Machado. On the influence of test smells on test coverage. In *Proceedings of the XXXIII Brazilian Symposium on Software Engineering*, pages 467–471, 2019.

[93] Sebastian Vogl, Sebastian Schweikl, Gordon Fraser, Andrea Arcuri, Jose Campos, and Annibale Panichella. Evosuite at the sbst 2021 tool competition. In *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*, pages 28–29. IEEE, 2021.

[94] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated, 2012.

[95] Aiko Yamashita and Leon Moonen. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)*, pages 306–315. IEEE, 2012.