2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)

Assessing Diffusion and Perception of Test Smells in Scala Projects

Jonas De Bleser, Dario Di Nucci, Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel, Brussels, Belgium
{jonas.de.bleser, dario.di.nucci, coen.de.roover}@vub.be

***Abstract*—Test smells are, analogously to code smells, defined as the characteristics exhibited by poorly designed unit tests. Their negative impact on test effectiveness, understanding, and maintenance has been demonstrated by several empirical studies.**

However, the scope of these studies has been limited mostly to JAVA in combination with the JUNIT testing framework. Results for other language and framework combinations are —despite their prevalence in practice— few and far between, which might skew our understanding of test smells. The combination of SCALA and SCALATEST, for instance, offers more comprehensive means for defining and reusing test fixtures, thereby possibly reducing the diffusion and perception of fixture-related test smells.

This paper therefore reports on two empirical studies conducted for this combination. In the first study, we analyse the tests of 164 open-source SCALA projects hosted on GITHUB for the diffusion of test smells. This required the transposition of their original definition to this new context, and the implementation of a tool (SOCRATES) for their automated detection. In the second study, we assess the perception and the ability of 14 SCALA developers to identify test smells. For this context, our results show (i) that test smells have a low diffusion across test classes, (ii) that the most frequently occurring test smells are LAZY TEST, EAGER TEST, and ASSERTION ROULETTE, and (iii) that many developers were able to perceive but not to identify the smells.

***Index Terms*—Test Smells, Test Quality, Scala Language**

I. INTRODUCTION

Automated unit testing [1], [2] has become a standard, but time-consuming activity in software engineering. Frameworks such as JUNIT ease the burden, but do not preclude developers from making suboptimal choices in the design of their tests. Indeed, test code should be just as understandable and maintainable as production code. Van Deursen *et al.* [3] defined test smells to this end, *i.e.,* characteristics of unit tests resulting from poor design choices, together with refactoring operations to eliminate them. Test code impacted by smells is hard to maintain and comprehend [4]. Moreover, recent studies have shown that they may also impact the deterministic behaviour of tests [5] and the quality of production code [6]. Despite the empirical evidence against test smells, developers tend not to be aware of the smells that exist in their tests [7] and can therefore benefit from tools that automate their detection.

The majority of the empirical research on test smells concerns the combination of JAVA and its unit test automation framework JUNIT. However, other programming languages have been introduced over the years and are being adopted in practice. SCALA [8], for instance, has enjoyed a steady rise in popularity over the past years —and for distributed systems in particular (see *e.g.,* [9], [10]). Some of their characteristics have enabled the design of unit test automation frameworks with sometimes unique features, which might impact our understanding of diffusion and test smells. The SCALATEST framework[1], for instance, leverages SCALA's unification of advanced object-oriented and functional features to offer a spectrum of 8 unit test definition styles ranging from the JUNIT-style "*FlatSpec*" familiar to JAVA developers, over the RSpec-style "*FunSpec*" familiar to Ruby developers, to the BDD-style "*FreeSpec*" that strives to resemble natural language specifications. Likewise, the framework leverages SCALA's support for first-class functions and traits to offer a comprehensive set of features for defining, re-using, and composing test fixtures in a fine-grained manner.[2] The combination of SCALA and the SCALATEST framework arguably gives developers more diverse options in the design and implementation of their unit tests than the combination of JAVA and the JUNIT framework. These newly-enabled test design options might impact the diffusion of test smells among real-world tests, as well as their perception by real-world developers.

To increase the subject diversity among the existing empirical studies on and strengthen the understanding of test smells, we investigate (i) the diffusion of test smells as well as (ii) the perception of test smells by developers in the SCALA ecosystem.

For the first empirical study, we reconsider the definitions of six test smells by Van Deursen et al. [3] in this new context, and present a tool for their automated detection in 164 open-source SCALA projects hosted on GITHUB. Quite surprisingly, the results show that these test smells have a low diffusion across test classes —with LAZY TEST, EAGER TEST, and ASSERTION ROULETTE as the most prevalent ones.

The goal of the second empirical study is to understand to what extent SCALA developers perceive existing smells in tests and consider them a severe issue. In particular, we survey 14 professional developers working on open-source and industrial SCALA software systems of varying size and scope. On average, we find that 5 out of 14 developers are able to identify (*i.e.,* perceive and explain) all the test smells. In summary, the contributions of this work are four-fold:

[1]http://www.scalatest.org
[2]http://www.scalatest.org/user_guide/sharing_fixtures

2574-3864/19/$31.00 ©2019 IEEE
DOI 10.1109/MSR.2019.00072

- The transposition of six test smells introduced by Van Deursen *et al.* [3] to the SCALA context, including the refactoring required to eliminate them, supported by a well-defined static method for their detection.
- The open-source SOCRATES (**SC**ala **RA**dar for **TE**st **S**mells) tool for automatically detecting these smells in SCALA projects.
- A large-scale empirical study that analyses the diffusion of test smells in the top-164 open-source SCALA projects hosted on GITHUB.
- A survey of 14 professional SCALA developers that assesses to what extent they are able to perceive and identify test smells in SCALA tests.
- A publicly available appendix [11] consisting of all data needed to replicate the studies or to conduct similar ones.

The remainder of the paper is organised as follows. Section II describes the test smells that we consider, the rules that we adopt to detect them in SCALA projects, and the corresponding refactorings. Sections III and IV describe our empirical studies including the research questions and the results that we obtained. Section VII discusses the threats that could affect the validity of the results. Section VI discusses the related literature. Finally, Section VIII concludes the paper.

II. TEST SMELLS IN SCALA SYSTEMS

We start our discourse by presenting our study subjects. Van Deursen *et al.* [3] originally introduced test smells using examples implemented in JAVA and the JUNIT framework. We now reconsider the former study in the context of SCALA and its SCALATEST framework. The choice for this framework is motivated by its distribution among the real-world projects in our corpus (cf. Figure 1 of Section III). As this is the first study on SCALATEST test smells, we focus our resources on the six smells considered by the majority of the existing empirical studies for JUNIT [4], [6], [12]–[15] —which includes the smells with the highest observed diffusion [13], [16].

For each test smell, we provide: (i) an adaption of its definition to the new context, (ii) an example instance, (iii) a static detection method based on the rules introduced by Bavota *et al.* [4], [12], and (iv) a refactoring to eliminate the smell and its negative impact. Note that all example tests share the same classes under test; `Ingredient` and `Recipe` depicted in Listing 1.

```scala
case class Ingredient(name: String, weight: Int)
case class Recipe(name: String, ingredients: List[Ingredient]) {
  def names: List[String] = ingredients.map(_.name)
  def hasIngredients: Boolean = ingredients.nonEmpty
}
object Recipe {
    def fromFile(file: BufferedSource): Recipe = ...
}
```

Listing 1. Example SCALA classes under test.

A. ASSERTION ROULETTE *(AR)*

Definition. A test case that contains more than one assertion of which at least one does not provide a reason for assertion failure. In case the test fails, this test smell encumbers identifying which assertion failed and the reason why. Listing 2 depicts a SCALA example of this test smell and its resolution.

Detection Method. The detection method for this test smell amounts to finding all assertions in a test case and verifying that each assertion is provided with an additional argument.

Refactoring. SCALATEST complements the familiar `assert` with the more expressive `assertResult`, `assertThrows`, `cancel`, `assume`, and `fail`. Each takes the assertion's failure explanation as an optional argument. The framework also provides the `withClue` construct which uses its given parameter as the failure explanation for all of the assertions in its scope. ASSERTION ROULETTE can therefore be resolved by either (i) providing a description or clue as an additional argument to `assert` and its variants, or by (ii) wrapping the assertions inside a `withClue`.

```scala
"A recipe with one ingredient" should "have names=List('Chocolate')" in {
   val recipe = Recipe("Chocolate Cookies", List(Ingredient("Chocolate", 100)))
   assert(recipe.names.head == "Chocolate")
   assert(recipe.names.size == 1)
}
```

```scala
"A recipe with one ingredient" should "have names=List('Chocolate')" in {
  val recipe = Recipe("Chocolate Cookies", List(Ingredient("Chocolate", 100)))

  assert(recipe.names.head == "Chocolate",
      s"The name of the ingredient was ${recipe.names.head}")

  withClue(s"The size of the 'names' was ${recipe.names.size}") {
    assert(recipe.names.size == 1)
  }
}
```

Listing 2. Example ASSERTION ROULETTE and its refactoring.

B. EAGER TEST *(ET)*

Definition. A test case that checks or uses more than one method of the class under test. Since its introduction [3], this smell has been somewhat broadly defined. It is left to interpretation which method calls count towards the maximum. Either all methods invoked on the class under test could count, or only the methods invoked on the same instance under test, or only the methods of which the return value is eventually used within an assertion. We have opted for the first interpretation in this study, but all others are valid too.

Detection Method. Our method to detect this smell consists of three steps: (i) identifying the class under test and collecting all of its methods; (ii) collecting the set of methods called from the test case; (iii) computing the size of the intersection of the outcomes (i) and (ii). If the intersection is larger than 1, more than one method is being tested by the test case.

Refactoring. Splitting the test into test cases that each test a single method of the class under test. For the example depicted in Listing 3, we opt to use a fixture to avoid duplicating the recipe object in each test case.

```scala
"The recipe" should "have two ingredients" in {
  val ingredients = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  val recipe = Recipe("Cookies and Milk", ingredients)
  assert(recipe.hasIngredients, "...")
  assert(recipe.names.equals(List("Cookie", "Milk")), "...")
}
```

```scala
def fixture = new {
  val ingredients = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  val recipe = Recipe("Cookies and Milk", ingredients)
}

"The recipe" should "have two ingredients" in {
  val f = fixture
  assert(f.recipe.names.equals(List("Cookie", "Milk")), "...")
}

"The recipe" should "have ingredients" in {
  val f = fixture
  assert(f.recipe.hasIngredients, "...")
}
```

Listing 3. Example EAGER TEST and its refactoring.

C. General Fixture *(GF)*

Definition. A test fixture that is too general. Ideally, test cases should use all the fields provided by their fixture. This might be difficult to uphold when the fixture is shared by several test cases. ScalaTest features no less than 4 different means for defining and sharing fixtures. The detection methods and refactorings for this smell are four-fold too.

Type I - Global Fixture: Detection Method. Similar to JUnit, ScalaTest supports defining fixtures by mixing in the trait `BeforeAndAfter` or `BeforeAndAfterEach` in a class. These traits respectively enable providing code, as the value for a by-name parameter to methods `before` or `after`, that must run before or after the test or each test case of the test. This code typically initializes the fields used within the test or test case. The detection of this smell requires three steps: (i) identify a test class that mixes in one of these traits and calls their `before` or `after` methods, (ii) collect the set of fields assigned in the code provided as an argument to these methods, and (iii) determine whether a test case of the class does not reference one of the assigned fields.

Type I - Global Fixture: Refactoring. The test cases defined in Listing 4 share none of the fields defined in their common fixture. This instance of the smell can be eliminated by removing trait `BeforeAndAfter` from the test class, and by demoting the fields referenced in the argument to method `before` to local, immutable variables in the appropriate test case. In case groups of test cases each use a different group of fields, and the groups should remain together, ScalaTest supports defining a local fixture per individual test case rather than a global fixture for the entire test class —which can be reused as illustrated in the remainder of this section.

```scala
class RecipeTestSuite extends FlatSpec with BeforeAndAfter {
  var emptyRecipe: Recipe = _
  var recipe: Recipe = _

  before {
    emptyRecipe = Recipe("Empty", List.empty[Ingredient])
    recipe = Recipe("Cookies and Milk",
    List(Ingredient("Cookie", 100), Ingredient("Milk", 200)))
  }

  "The recipe" should "have two ingredients" in {
    assert(recipe.names.equals(List("Cookie", "Milk")), "...")
  }

  "The empty recipe" should "have no ingredients" in {
    assert(!emptyRecipe.hasIngredients, "...")
  }
}
```

```scala
class RecipeTestSuite extends FlatSpec {
  "The recipe" should "have two ingredients" in {
    val ingredients =  List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
    val recipe = Recipe("Cookies and Milk", ingredients)
    assert(recipe.names.equals(List("Cookie", "Milk")), "...")
  }

  "The empty recipe" should "have no ingredients" in {
    val emptyRecipe = Recipe("Empty", List.empty[Ingredient])
    assert(!emptyRecipe.hasIngredients, "...")
  }
}
```

Listing 4. Example Type I General Fixture and its refactoring.

Type II - Loan Fixture: Detection Method. So-called "loan fixture methods" are methods with a body that serves to set up and tear down fixture objects, respectively before and after the call from their body to the function provided to them as a parameter. Method `withRecipe` in Listing 5 is such a loan fixture method, calling its parameter `test` on line 8 with the fixture objects it has set up. The method itself is called from line 11 and line 16, for the purpose of loaning the objects to the test cases defined by its function argument on lines 12–13 and lines 17–18 respectively. Multiple loan fixture methods can be defined in a test class, and shared with the appropriate test cases. Despite the increase in expressivity, this definition style is not less prone to the GF test smell. Detecting the GF smell in fixtures defined through loan fixture methods requires: (i) collecting the parameters of the function given as an argument for the call to the loan fixture method from the test case, and (ii) checking whether every parameter is referenced in the body of the function.

Type II - Loan Fixture: Refactoring. The fixture should be removed, in case the test case uses none of its objects, or split into several local fixtures. Note the changes in the parameter and argument lists as a result of the refactoring below, as well as the composition of two separate local fixtures for the last test case.

```scala
1  class RecipeTestSuiteLF extends FlatSpec {
2
3    def withRecipe(test: (Recipe, Recipe) => Any) {
4      val ingredients1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
5      val ingredients2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
6      val cookiesAndMilk = Recipe("Cookies and Milk", ingredients1)
7      val baconAndEggs = Recipe("Eggs", ingredients2)
8      test(cookiesAndMilk, baconAndEggs)
9    }
10
11   "The recipe" should "have 2 ingredients (Eggs, Bacon)" in withRecipe {
12     (cookiesAndMilk, baconAndEggs) =>
13       assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
14   }
15
16   "The recipe" should "have 2 ingredients" in withRecipe {
17     (cookiesAndMilk, baconAndEggs) =>
18       assert(cookiesAndMilk.ingredients.size == 2, "...")
19   }
20 }
```

```scala
1  class RecipeTestSuite extends FlatSpec {
2
3    def withCookiesAndMilk(test: (Recipe) => Any) {
4      val ingredients = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
5      val cookiesAndMilk = Recipe("Cookies and Milk", ingredients)
6      test(cookiesAndMilk)
7    }
8
9    def withBaconAndEggs(test: (Recipe) => Any) {
10     val ingredients = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
11     val baconAndEggs = Recipe("Eggs", ingredients)
12     test(baconAndEggs)
13   }
14
15   "The recipe" should "have 2 ingredients (Eggs, Bacon)" in withBaconAndEggs {
16     baconAndEggs =>
17       assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
18   }
19
20   "The recipe" should "have 2 ingredients" in withCookiesAndMilk {
21     cookiesAndMilk =>
22       assert(cookiesAndMilk.ingredients.size == 2, "...")
23   }
24
25   "Different recipes" should " not be equal" in withBaconAndEggs {
26     baconAndEggs =>
27       withCookiesAndMilk { cookiesAndMilk =>
28         assert(cookiesAndMilk.equals(baconAndEggs), "...")
29       }
30   }
31 }
```

Listing 5. Example Type II General Fixture and its refactoring.

Type III - Fixture Context: Detection Method. So-called "fixture context" objects are instances, such as the ones instantiated on lines 11–13 and lines 15–17 of Listing 6, of an anonymous class that mixes in at least one trait such as `RecipeFixture` that provides *and* initializes fields for the fixture. The body of the anonymous class itself corresponds to the test case, such as the `assert` expressions on lines 12 and 16. Note that multiple traits can be mixed into the "fixture context" object (*e.g.,* `new X with Y with Z`) as required by the fixture for a specific test case. Fixtures defined in this manner, as expressive it may be, are still prone to the GF test smell. Its detection requires: (i) collecting the fields mixed into and provided by the “fixture context” object, (ii) verifying whether every field is referenced in test case (*i.e.,* the body of the corresponding anonymous class creation expression).

Type III - Fixture Context: Refactoring. The refactoring consists of splitting the fixture into multiple smaller fixtures. A trait can be dedicated to each field, rendering them easier to compose as needed for individual test cases.

```scala
class RecipeTestSuiteFCO extends FlatSpec {

  trait RecipeFixture {
    val ingredients1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
    val ingredients2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
    val cookiesAndMilk = Recipe("Cookies and Milk", ingredients1)
    val baconAndEggs = Recipe("Eggs", ingredients2)
  }

  "The recipe" should "have two ingredients (Eggs, Bacon)"
    in new RecipeFixture {
      assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
  }

  "The recipe" should "have two ingredients" in new RecipeFixture {
    assert(cookiesAndMilk.ingredients.size == 2, "...")
  }
}
```

```scala
class RecipeTestSuiteFCOR extends FlatSpec {

  trait BaconAndEggsRecipe {
    val baconAndEggs = Recipe("Eggs", List(Ingredient("Eggs", 100),
      Ingredient("Bacon", 200)))
  }

  "The recipe" should "have two ingredients named Eggs and Bacon"
      in new BaconAndEggsRecipe {
          assert(baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
    }
}
```

Listing 6. Example Type III General Fixture and its refactoring.

Type IV - With Fixture: Detection Method. One more fixture definition style is available to classes that extend a type from package `org.scalatest.fixture`. Each of the test cases in such a class take the same fixture as parameter, such as `f` on line 13 of Listing 7. This fixture can be set up and torn down by overriding method `withFixture` in the test class, the body of which needs to apply the method’s function parameter —which corresponds to the executed test case— to the fixture. This definition style eliminates some of the boilerplate involved in the “loan fixture method” style, but is only applicable when most test cases share the same fixture. The GF smell can manifest itself if the class defining the fixture (*e.g.,* `FixtureParam` on line 3) provides fields that are not referenced from a test case. It is convenient and common to use the `case class` feature of Scala to define the fixture class, which is the only variant we can support detecting without computationally expensive program analyses. The detection requires: (i) finding test classes that inherit from package `org.scalatest.fixture`, (ii) resolving the type of the argument to the function called from within `withFixture` to its type definition, and (iii) ensuring that all test cases within the class use the fields provided by this case class.

Type IV - With Fixture: Refactoring. There are multiple ways to eliminate this smell, but it is clear that the other definition styles such as Loan Fixture or Fixture Context can be of help. As an example, please refer to Listing 5 which represents a potential refactoring.

```scala
1  class RecipeTestSuiteWF extends fixture.FlatSpec {
2
3    case class FixtureParam(cookiesAndMilk: Recipe, baconAndEggs: Recipe)
4
5    def withFixture(test: OneArgTest): Outcome = {
6      val ingredients1 = List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
7      val cookiesAndMilk = Recipe("Cookies and Milk", ingredients1)
8      val ingredients2 = List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
9      val baconAndEggs = Recipe("Eggs", ingredients2)
10     val theFixture = FixtureParam(cookiesAndMilk, baconAndEggs)
11     test(theFixture)
12   }
13
14   "The recipe" should "have two ingredients named Eggs and Bacon" in { f =>
15     assert(f.baconAndEggs.names.equals(List("Eggs", "Bacon")), "...")
16   }
17
18   "The recipe" should "have two ingredients" in { f =>
19     assert(f.cookiesAndMilk.ingredients.size == 2, "...")
20   }
21 }
```

Listing 7. Example Type-IV General Fixture.

D. Lazy Test *(LT)*

Definition. More than one test case with the same fixture that tests the same method. This smell affects test maintainability, as assertions testing the same method should be in the same test case. Like Eager Test, the original definition [3] leaves some details to interpretation. We consider every call to the class under test as a potential cause of Lazy Test, irrespective of whether their results are used in an assertion.

Detection Method. A Lazy Test can be detected in three steps: (i) identify the class under test and collect its methods, (ii) collect the set of method calls to the class under test in each test case, (iii) compute the size of the intersection of the outcomes of (i) and (ii). A non-empty intersection is indicative of a method tested by multiple test cases.

Refactoring. Merge the individual test cases that execute the same method into a single one. The result is one test case per method of the class under test, which is said to improve the traceability between production and test code.

```scala
"The recipe" should "have zero ingredients" in {
  val recipe = Recipe("Cookies and Milk", List.empty)
  assert(
    !recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}"
  )
}

"The recipe" should "have two ingredients" in {
  val recipe = Recipe(
    "Cookies and Milk",
    List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  )

  assert(
    recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}"
  )
}
```

```scala
"The recipe" should "zero ingredients" in {
  val emptyRecipe = Recipe("Cookies and Milk", List.empty)
  val recipe = Recipe(
    "Cookies and Milk",
    List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
  )

  assert(
    !emptyRecipe.hasIngredients,
    s"The number of ingredients was ${emptyRecipe.ingredients.size}"
  )

  assert(
    recipe.hasIngredients,
    s"The number of ingredients was ${recipe.ingredients.size}"
  )
}
```

Listing 8. Example Lazy Test and its refactoring.

E. Mystery Guest *(MG)*

Definition. A test case that uses external resources that are not managed by a fixture. A drawback of this approach is that the interface to external resources might change over time necessitating an update of the test case, or that those resources might not be available when the test case is run, endangering the deterministic behavior of the test. Note that we do not consider RESOURCE OPTIMISM [3], which requires analysing the state or the existence of external resources. Indeed, RESOURCE OPTIMISM is a special case of MYSTERY GUEST and the following refactoring resolves both smells.

Detection Method. This smell can be detected by identifying test cases that contain instance creation expressions for resource-related types (*e.g.,* `java.io.File`, `java.nio.file.Path`,`java.io.FileInputStream`, or `java.net.URI`) and factory methods for reading files such as `scala.io.Source#fromFile`, or methods for setting up connections to a database using `java.sql.DriverManager#getConnection`.

Refactoring. Manage resources explicitly in a fixture.

```
"A recipe" should "be able to be initialized from a file" in {
  val file = scala.io.Source.fromFile("ingredients_recipe.txt")
  val recipe = Recipe.fromFile(file)
    assert(recipe.ingredients.size == 20,  "...")
}
```

```
def withRecipeFile(test: BufferedSource => Any) {
  val path = "file.txt"
  val contents =
    """
    |BaconAndEggs
    |bacon,100
    |eggs, 200
    """.stripMargin

  Files.write(Paths.get(path), contents.getBytes(StandardCharsets.UTF_8))
  assume(new File(path).exists(), s"File $path did not exists")
  test(scala.io.Source.fromFile(path))
}

"A recipe" should "be able to be created from file" in withRecipeFile { file =>
  val recipe = Recipe.fromFile(file)
  assert(recipe.ingredients.size == 20,  "...")
}
```

Listing 9. Example MYSTERY GUEST and its refactoring.

F. SENSITIVE EQUALITY *(SE)*

Definition. A test case with an assertion that compares the state of objects by means of their textual representation, *i.e.,* by means of the result of `toString()`.

Detection Method. This smell can be detected by (i) enumerating the assertions in a test case (as described in section II-A) and by (ii) verifying whether they contain or rely on a call to the `toString()` method.

Refactoring Compare the members of the object states structurally instead of relying on `toString()` of the wholes.

```
"A recipe" should "have as ingredient: Ingredient('Chocolate', 100)" in {
  val recipe = Recipe("Chocolate Cookies", List(Ingredient("Chocolate", 100)))
  assert(recipe.toString() == "Recipe(Chocolate Cookies,List(Ingredient(
    Chocolate,100)))", "...")
}
```

```
"A recipe" should "have as ingredient: Ingredient('Chocolate', 100)" in {
  val recipe = Recipe("Chocolate Cookies", List(Ingredient("Chocolate", 100)))
  assert(recipe.ingredients == List(Ingredient("Chocolate", 100)), "...")
}
```

Listing 10. Example SENSITIVE EQUALITY and its refactoring.

III. DIFFUSION OF TEST SMELLS

We now present the design and the results of our empirical study on test smell diffusion in open-source SCALA systems.

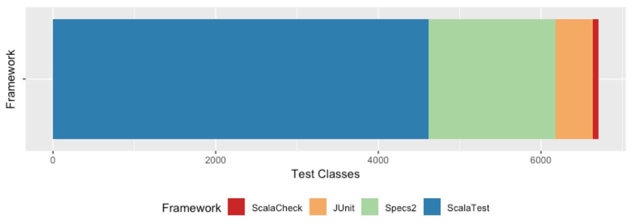

Figure 1. Usage distribution of unit testing frameworks in SCALA systems.

A. Study Design

The *goal* of our study is to analyse test smell diffusion in open-source SCALA systems using the detection methods defined in Section II, for the *purpose* of increasing the subject diversity among the existing empirical studies on test smells, and understanding whether SCALA's testing frameworks — with their unique feature sets and support for multiple test and fixture definition styles— impact diffusion, from the *perspective* of researchers in and tool builders for software quality. The study aims to answer the following research questions:

- **RQ$_1$:** *To what extent are test smells spread across test classes written in* SCALA*?*
- **RQ$_2$:** *Which test smells occur more frequently in test classes written in* SCALA*?*

The initial dataset consists of $72,619$ open-source SCALA projects that were created on Github between January 2010 and July 2018. We discarded the projects that (i) lack test classes, (ii) are not using the SCALA BUILD TOOL[3] (SBT) for build automation (required to obtain semantic information for smell detection), or (iii) are outdated and no longer compile. After this preprocessing phase, $2,920$ projects remain.

Based on this dataset, we conduct a preliminary analysis to assess which of the testing frameworks SCALATEST, SPECS2, SCALACHECK and JUNIT are the most prevalent. The resulting usage distribution for each framework is shown in Figure 1. It is clear that SCALATEST is the most used framework. Surprisingly, the popular JUNIT framework is barely used, despite the interoperability of JAVA and SCALA.

The remainder of our study focuses on projects that use SCALATEST, the most widely used testing framework, and that have more than $1,000$ LOC in both production and test code. Thus, the final dataset is composed of 164 projects, summarized in Table I. Given its size, a manual inspection of the test classes for test smells is prohibitive. We developed SOCRATES to this end, which supports all 8 test definition styles of SCALATEST —of which the "*FlatSpec*", "*FunSuite*", and "*WordSpec*" styles are the most prevalent among the test classes in our dataset, as shown in Table III. For every project in the dataset, the SOCRATES tool:

1) clones the master branch from the GITHUB repository, and verifies that the project uses SBT for build automation;

[3]https://github.com/sbt/sbt

TABLE I
DATASET CHARACTERISTICS OF 164 SCALA PROJECTS.

	1st Quartile	Mean	Median	3rd Quartile	Total
# of Production Files	25.75	74.79	48.00	86.00	12,266
# of Test Files	13.75	35.62	20.50	42.25	5,841
# of Production LOC	2,107.25	7,236.02	3,717.50	6,740.25	1,186,708
# of Test LOC	1,400.00	3,958.37	1,959.5	4,032.75	649,172
# of Test Classes	9.75	29.96	15.50	31.00	4,914
# of Test Cases	49.75	149.87	84.50	184.75	24,578

TABLE II
PERCENTAGE OF PROJECTS, TEST CLASSES, AND TEST CASES EXHIBITING THE DIFFERENT SMELLS ALONG WITH PRECISION AND RECALL OF OUR TOOL FOR EACH SMELL.

Test Smell	% per Projects	% per Test Class	% per Test Case	Precision	Recall
AR	44.51%	15.97%	12.71%	100.00%	100.00%
ET	51.82%	6.57%	. 6.12%	96.49%	66.27%
GF - Type I	10.36%	1.11%	1.22%	96.67%	96.67%
GF - Type II	5.48%	0.38%	0.13%	-	-
GF - Type III	9.75%	1.62%	1.53%	100.00%	88.87%
GF - Type IV	1.82%	0.28%	0.18%	-	-
LT	62.19%	11.05%	22.99%	99.44%	75.32%
MG	15.85%	1.95%	1.41%	100.00%	100.00%
SE	13.41%	2.72%	1.12%	100.00%	100.00%

2) compiles the project using the SEMANTICDB[4] compiler plugin for SBT which exposes the semantic information maintained by the compiler (*i.e.,* symbol and type resolution),
3) parses the test classes to Abstract Syntax Trees (AST's),
4) determines the used testing style for each test class,
5) collects all test cases within the test class, and
6) uses the AST of each test case *and* the extracted semantic information to detect the presence of test smells using the methods defined in Section II.

B. Analysis of the Study Results

Table II lists, for each test smell, the percentage of projects and the percentage of test classes that exhibit at least one instance of that smell. Figure 2 depicts the diffusion of test smells on a per-project basis (*i.e.,* for each project, the percentage of test classes that exhibit a test smell). For the actual answer to **RQ$_1$**, we investigated how many test classes and projects are affected by at least one test smell. We find that $1,381$ out of $4,914$ (28.10$\%$) test classes and that 138 out of 164 (84.14%) projects are affected by at least one test smell. These numbers differ from those found by the studies that target the combination of JAVA and JUNIT (*i.e.,* [12], [16]), which we discuss in section V.

Summary for RQ$_1$. Although 84.14% of the projects are affected by at least one test smell, only 28.10% of their test classes exhibit them. Therefore, we conclude that the diffusion of test smells in SCALA projects is not very high with respect to the one observed in JAVA projects [4], [12].

Turning to **RQ$_2$**, it is clear that LAZY TEST, EAGER TEST, and ASSERTION ROULETTE are the most prevalent test smells. LAZY TEST occurs in almost 2 out of 3 projects (62.19%), or in 11.05% of the test classes. Thus, many projects have at least one test class that is affected by this smell, but the smell does not occur frequently in multiple test classes. EAGER TEST occurs in about 1 out of 2 projects (51.82%), but only in 6.57% of the test classes. Thus, we observe that instances of EAGER TEST do not occur frequently in several test classes. Finally, ASSERTION ROULETTE occurs in almost half of projects (44.51%) and in 15.97% of the test classes. On average, LAZY TEST, ASSERTION ROULETTE, and EAGER TEST occur in half of the SCALA systems.

[4]https://scalameta.org/docs/semanticdb/guide.html

TABLE III
DISTRIBUTION OF TEST DEFINITION STYLES AMONG THE TEST CLASSES.

FlatSpec	FunSuite	WordSpec	FunSpec	FreeSpec	FeatureSpec	PropSpec	RefSpec
46.47%	26.73%	11.41%	10.66%	3.64%	0.59%	0.46%	0%

TABLE IV
DISTRIBUTION OF TEST FIXTURE DEFINITION STYLES AMONG THE TEST CASES WITH A FIXTURE.

Type I - Global	Type II - Loan Method	Type III - Context Object	Type IV - With
44.85%	17.06%	33.87%	4.20%

The remaining three smells GENERAL FIXTURE, MYSTERY GUEST, and SENSITIVE EQUALITY are less diffused. GENERAL FIXTURE occurs in about 1 out of 4 projects (27.41%) and in 3.39% of the test classes. Note that these results are the sum of the results for the four types of GENERAL FIXTURE, as explained in Section II. In contrast to JUNIT, where the fixture of a test class applies to all of its test cases by default, the first-class status of traits and functions in SCALA enables SCALATEST to support the fine-grained definition and sharing of fixtures between individual test cases through so-called "fixture context objects" and "loan fixture methods" respectively. This could have a positive impact on the prevalence of the GENERAL FIXTURE test smell. Indeed, the distribution in Table IV shows that about 51% of all test cases with a fixture use one of these fixture definition styles.

MYSTERY GUEST is present in about 1 out of 7 (15.85%) projects, but in only 1.95% of all test classes. The last smell, SENSITIVE EQUALITY, is present in 13.41% and 2.72% of the projects and test classes respectively.

Summary for RQ$_2$. Across SCALA projects, LAZY TEST (62.19%), EAGER TEST (51.82%), and ASSERTION ROULETTE (44.51%) are the three most prevalent test smells, while GENERAL FIXTURE (27.41%), MYSTERY GUEST (15.85%), and SENSITIVE EQUALITY (13.41%) are the least prevalent.

It is important to note that the SOCRATES tool does not only use syntactic information from the ASTs of the test cases, but also semantic information such as types and definition-use relations from their compilation. This enables detecting the aforementioned test smells more precisely (Table II), but might come at the expense of recall. We therefore investigated the precision and recall of SOCRATES *manually* by validating a statistically significant sample (confidence level: 95%, confidence interval: 5%) of the test cases. Two of the authors inspected 377 test cases to determine whether the tool correctly identified or missed the smells in each test case. Disagreements

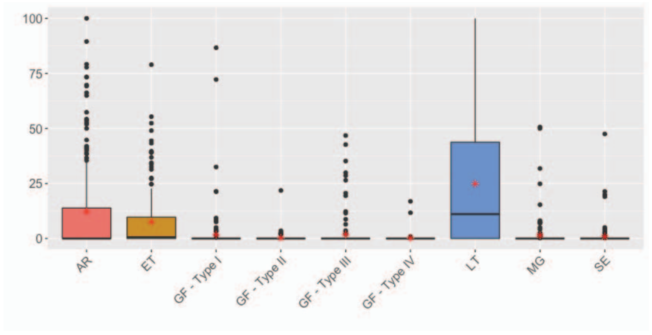

Figure 2. Diffusion of test smells across SCALA Projects

during the validation were resolved by carefully checking both the code snippet and the test smell definition. The last two columns of Table II list the precision and recall of the tool for each type of test smell, while the complete results are available in our appendix [11]. Across the smells, SOCRATES achieves a high precision of 98.94% and a more than modest recall of 89.59%. These values are similar to those obtained by the tools used in previous empirical studies such as the one performed by Palomba *et al.* [16] that achieved 88% of precision and 100% of recall. Therefore, we deem the tool sufficiently strong to support our conclusions.

IV. PERCEPTION OF TEST SMELLS

We now describe the empirical study into the extent to which SCALA developers perceive test smells as problematic.

A. Study Design

The *goal* of the study is assess the knowledge of test smells among and the perception of their severity by SCALA developers, for the *purpose* of increasing the subject diversity among the studies into this matter, prioritizing research on and tool support for the identification and elimination of specific smells, and assessing the need for knowledge dissemination on test smells —from the *perspective* of researchers in, tool builders for, and educators of software quality. The study aims to answer the following research question:

- **RQ$_3$:** *To what extent do developers perceive smelly tests and are able to identify the actual smells in test classes written in* SCALA*?*

The study takes the form of a survey among professional SCALA developers.

Objects. We consider the test smells discussed in Section II as the objects of this survey. At the beginning of every survey question, we show the code for the classes under test from Listing 1. For the test smell in question, we then provide the corresponding example unit test from Section II that exhibits this smell, without providing any additional information about the design flaw. As such, there is no need to explain the context and purpose of the system under test further. Each participant has to indicate whether he or she perceives the test smell and has to motivate their answer through the following questions:

- *Does this unit test exhibit a test smell according to your experience?*
- *If yes, indicate which piece(s) of code and/or which reasons might cause this test smell. If no, leave blank.*
- *If yes, how severe would you rate this test smell on a Likert scale. If no, leave blank.*

Participants. To have a representative sample of SCALA developers, we invite members of several SCALA meet-up groups[5] to participate to our survey. In total, 14 developers were willing to participate in our survey. All are professionals with most developers (10 out of 14) working on industrial systems, and the remaining (4 out 14) working on open-source systems. 13 participants have more than 5 years of experience developing in SCALA, while 10 have more than 10 years of experience. This is impressive given that SCALA was released only in 2004. They use multiple testing frameworks: SCALATEST (78.6%), SCALACHECK (50%), SPECS2 (42.9%), and JUNIT (21.4%). This partially confirms the analysis on the most popular testing frameworks presented in Section III. 11 of the participants consider themselves as experienced in the domain of software testing and 9 of them as experienced SCALA developers. Their experience is also reflected by the size of the biggest project on which they have worked. Indeed, most projects range from 10K to 100K LOC. The survey is hosted on Google Forms and is designed to be completed in approximately 20 minutes. We collect the answers over a timespan of 2 weeks.

We address **RQ$_3$** by means of Figure 3. This plot depicts the number of participants that perceived and identified each test smell. More specifically, it depicts:

- The absolute number of cases in which test smells have been *perceived* by participants. A test smell is perceived whenever the participant answered yes to the question: *Does this unit test exhibit a test smell according to your experience?*
- The absolute number of cases in which test smells have been *identified* by the participants. A participant is able to identify a test smell if the given explanation correctly pinpoints the cause of the test smell. The explanations

[5]https://www.meetup.com

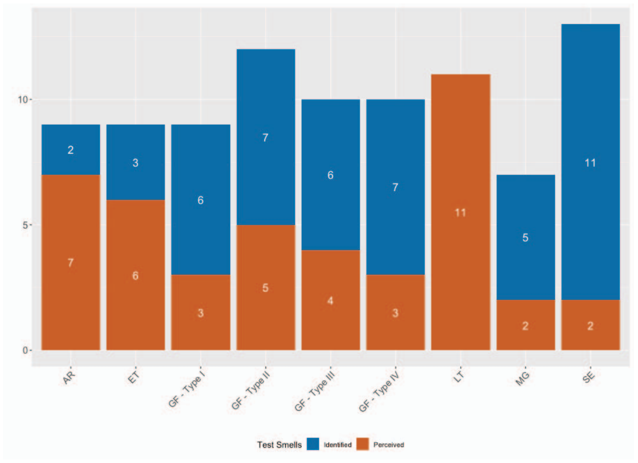

Figure 3. Absolute number of participants that perceived the smells compared to the ones that correctly identified them.

are collected from the question *"If yes, indicate which piece(s) of code and/or which reasons might cause this test smell. If no, leave blank."*. For example, an explanation that clearly verifies that the participant pinpointed GENERAL FIXTURE is the following: *"[...] concrete test data is best constructed and tailored for each specific tests."*, while one such as *"Too much scaffolding"* is considered too general and thus we cannot assert that the participant correctly identified the test smell.

Based on the aforementioned data analysis, we are able to verify which test smells are the most perceived and which ones are most identified (**RQ**$_3$).

B. Analysis of the Results

For **RQ**$_3$, we analyse the survey answers quantitatively and quantitatively.

Assertion Roulette. This smell was perceived by 9 out of 14 participants. However, only 2 of them identify the problem of the test having multiple assertions without failure explanation (*e.g., [...] several assertions within same test case*).

General Fixture. We discuss all variants of the GENERAL FIXTURE smell. We observe that, on average, 10 out of 14 participants perceive each of the variants. Additionally, 7 out of 14 participants also correctly identify the cause of these smells. It emerges that developers are well aware of the correct usage of fixtures, as confirmed by several detailed explanations of the participants such as: *"Two different test cases uses the same fixture that is badly tailored for each. Each test case should have specific data made for it, reusing test data should remove code duplication when needed, in this case it's coupling the scenarios for no reason"* and *"unused fixtures in tests, plus in this case because each fixture is only used in one test you're paying a readability penalty for nothing"*.

Eager Test. EAGER TEST smell is perceived by 9 out of 14 participants, but only 3 of them are able to correctly identify the cause of this smell. In these cases, the explanations are very short and to the point, such as *"Should only test one thing, and [...]"*.

Mystery Guest. Half of the participants (7 out of 14) perceive that the unit test exhibited an instance of MYSTERY GUEST. 5 of these 7 developers identify the problem (*e.g., "[...] instead of a file would have simplified testing to avoid using external resources, there are also alternatives to generate tmp files for the test case, so that the file content and the test checks are easier to keep in sync"*).

Sensitive Equality. Almost all participants (11 out of 14) perceive the smell. They correctly identify the problematic piece of code, namely the use of `toString`. Their explanations are all very similar and correctly pinpoint the problem. One participant, for instance, does so as follows: *"Testing 'toString' to check for behaviour is easily broken. Adding parameters to the class or renaming it would immediately break the test."*. This is testament to the awareness of the smell.

Lazy Test. 11 out of 14 participants perceived this smell, but none was able to identify it correctly. It looks like many developers are simply not aware of this smell, which is also reflected by the large number of occurrences of this smell, as depicted in Figure 2. It is worth mentioning that one of the participants misinterpreted the name of the smell, and assumes that it is related to an incongruence between the test description (i.e., lines 1 and 9 of Listing 8) and its implementation: *"[...] you call it "the lazy test" but yeah, those assertions don't match the test labels, they aren't specific/strong enough assertions"*. Surprisingly, multiple participants remarked the same incongruence: *"Should only test one thing, and description should match assertion*, *"[...] Moreover, the description doesn't match the test: just verifying that [...]*, and *"The tests description does not match the assertions"* Its inclusion in the survey was unintentional, yet several developers perceive it as a new kind of test smell. This indicates that SCALATEST users rely on the descriptions of a test to understand its purpose. Note that JUNIT only recently provides the `@DisplayName` annotation to the same end.

Summary for RQ$_3$. The considered test smells are perceived by developers. The most identified test smells are SENSITIVE EQUALITY, GENERAL FIXTURE, and MYSTERY GUEST, while no developer was able to identify LAZY TEST correctly. Only 5 out of 14 (35.71%) are able to explain the cause of these smells. This shows that many developers are not able to correctly identify most of the smells, even though they perceive a design issue.

V. DISCUSSION

Given that SCALA and JAVA have the basics of object-oriented programming on the JVM in common, we performed a high-level discussion and comparison with the results obtained by previous studies that target test smells in JAVA systems. We did not perform a strict quantitative comparison because the datasets considered in the studies have different characteristics.

Diffusion. Greiler *et al.* [14] analysed the diffusion of fixture-related test smells, including the GENERAL FIXTURE studied in this paper, in 3 industrial JAVA systems that use either the JUNIT or TESTNG unit testing frameworks. They found that GENERAL FIXTURE occurs in respectively 13.5%, 23%, and 32% of each of the systems' test cases. Among the 164 open-source SCALA systems studied in this paper, we found an instance of GENERAL FIXTURE in only 3.06% of the test cases. Note that the studies differ on the detection method used for the smell in that SoCRATES reports the smell as soon as at least one field is not used by a test case, while Greiler *et al.* evaluate the ratio of referenced fixture fields against a 0.7 threshold —and still observe a higher diffusion. We find a potential explanation in an observation made by the authors in their discussion of fixture definition means: "The more fine-grained directives which TESTNG offers are not used". This is in contrast to the distribution among the SCALA test cases with a fixture depicted in Table IV. We therefore recommend that unit testing frameworks include features that support the fine-grained definition and sharing of case-specific fixtures, and that developers use them.

Bavota *et al.* [12] investigated the diffusion of test smells in 27 JAVA systems. We started from the detection methods outlined in their work for our transposition to the SCALA context in Section II. Still, they found 82% of the subject test classes to be affected by at least one test smell, which is higher than our 28.10% of test classes among the 164 SCALA systems. Part of the explanation must be their inclusion of the TEST CODE DUPLICATION and INDIRECT TESTING smells, affecting 35% and 11% of the JUNIT test classes respectively. However, the results for ASSERTION ROULETTE and EAGER TEST differ too: these smells occur in 55% and 34% respectively of the JUNIT test classes, and only in 15.97% and 6.57% of the SCALATEST test classes. The studies do agree on ASSERTION ROULETTE and EAGER TEST being among the three most prevalent test smells.

An explanation for the lower diffusion of EAGER TEST might be that fields are by default public in SCALA, and that its syntax enables protecting these fields at a later point in time by true accessor methods (*i.e.,* that do not directly return or set the value of the field) without having to substitute method calls for field accesses.[6] For the JUNIT study, calls to accessor methods —including those returning or setting the field's value directly— still count towards those considered in the EAGER TEST (and LAZY TEST) detection rule. We recommend researchers to exclude them instead.

An explanation for the lower diffusion of ASSERTION ROULETTE among the SCALATEST classes might be that the framework features a popular DSL for specifying assertions as `should`-based sentences such as `string should startWith regex "Hel*o"` or `a [Exception] should be thrownBy`, for which the framework does not take an explicit failure explanation message as argument but rather generates one itself based on the other operators in the sentence. Our analysis therefore does not consider them as an assertion without explanation. Similar DSLs might not be as popular or comprehensive for JUNIT, but we recommend their use from the perspective of co-evolution of assertion and explanation.

[6]A pair of getter `field` and setter `field =` methods can be defined so that existing `field` read and writes become calls to the appropriate method.

Perception. We are only aware of three other studies that assess the perception of test smells by developers.

Bavota *et al.* [12] complement the aforementioned diffusion results with a controlled experiment involving 61 participants ranging from students to professional developers. However, their main aim was not to assess the participants' perception of test smells, but to verify the impact of JUNIT smells on software maintenance. The results of the user study show that test smells negatively impact program comprehension during maintenance activities.

The aforementioned study by Greiler *et al.* [14] on fixture-related smells in JUNIT and TESTNG tests does include a survey among 13 professionals. The results show that developers recognise that fixture-related smells are problematic and agree that they impact test maintenance negatively. However, it is not clear whether developers would be able to recognize the smells without the detection tool used in the study.

The study with results closest to our own is by Tufano *et al.* [13] who investigated developers' perception of JUNIT test smells in a survey among 19 participants. Their study considers the same test smells, except for the omission of LAZY TEST, and found that developers do not really perceive test smells as actual design problems (only in 17% of the cases) and are even less capable of identifying them precisely (only in 2% of the cases) without tool support. The difference to our experimental design, apart from the focus on the combination of SCALA and SCALATEST, is that the tests used in the survey stem from real-world systems and that the participants are contributors to those projects. While these participants may be more aware of the design tradeoffs made in the system's history, they may also recognize its shortcomings less. The overall conclusions of both surveys align though.

VI. RELATED WORK

In the last decade, several methods and tools to detect design flaws in production code have been proposed [17]–[20], as well as empirical studies aimed at assessing their impact on maintainability [21]–[31].

Despite the importance of testing [1] and of well-designed test components [32], design problems affecting test code have been only partially explored. In this context, Van Deursen *et al.* [3] defined a catalog of 11 test smells, *i.e.,* a set of a poor design solutions to write tests, together with refactoring operations able to remove them. These smells take into account poor design choices adopted by developers during the implementation of test fixtures or of single test cases. Based on the work of Van Deursen *et al.* [3], another catalogue was defined by Meszaros [33].

Most of the approaches for test smell detection [14], [34] are based on structural metrics. Palomba *et al.* [35] argued that this information does not suffice for test smell detection and proposed TASTE, a text-based detector exploiting information retrieval techniques to identify test smells such as GENERAL FIXTURE and EAGER TEST. Their empirical study showed that exploiting textual information can lead to more accurate detectors. Later on, Spadini *et al.* [6] studied the relation of test smells and software quality. Their results show that test smells are an important problem for maintainability and reliability of software systems, badly influencing the quality of both production and test code. Similar findings were reported by Palomba and Zaidman [5], who discovered that three test smells can induce tests flakiness. Finally, Tufano *et al.* [13] highlighted that test smells are generally introduced when test code is committed for the first time, and often developers do not refactor them.

With respect to the papers described so far, our work is, to the best of our knowledge, the first one targeting SCALA, a more modern programming language with a rich ecosystem of testing frameworks, and of which the adoption has been rapidly increasing over the last decade.

VII. THREATS TO VALIDITY

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

Threats to construct validity. These threats mainly concern the measurements we performed. We detected test smells in a corpus of open-source SCALA projects using our SOCRATES static analysis tool. As its precision and recall have a major influence on the results, we manually validated SOCRATES in Section III-B on a statistically significant subset of test cases (377 test cases, with a confidence level of 95% and a confidence interval of 5%), and observed a precision of 98.94% and a recall of 89.59% —in line with those achieved by other state-of-the-art tools [4], [16].

We also note that some definitions of test smells leave details open to interpretation, while others require semantic information to detect them correctly. With respect to the former, we started our transposition to SCALA from the test smell detection rules used in the studies by Bavota *et al.* [4], [12], the results of which we compare against in the discussion section. With respect to the latter, our future work includes incorporating more of the semantic information provided by the SEMANTICDB library, and in particular information about the type hierarchy and call graph. We currently identify the corresponding production class for a test class through naming conventions such as those used in existing studies [4], [6], [16]. We acknowledge that SOCRATES might therefore have missed some links between production and test classes. Finally, we assume that `src/test` only contains unit tests and not integration tests as they should be in `src/it`. However, this is a convention that developers might not follow and it could bias the results.

A threat to the validity of the survey results is the artificial nature of the test smell samples that we provided to the participants. These samples were designed to focus developers on the design issues of the tests, and to avoid long training sessions about the systems under tests which are difficult to realize for studies with industrial developers. We are also aware that only showing examples of tests that exhibit a smell, along with its name, could have biased the participants. However, the impact is limited to the perception results only, as participants had to explain their answer for the identification results.

Threats to internal validity. These threats concern factors internal to our study that we cannot control. The number of participants that we found willing to participate in the survey represents the largest of these threats for its results. We interviewed 14 experienced SCALA developers working on open-source and industrial software systems. This sample is by no means representative and replications with a larger number of participants are desirable.

Threats to external validity. These threats concern the generalization of our results. For the study on the diffusion of test smells, we selected a subset composed of the top-164 SCALA projects hosted on GITHUB. We considered only projects with test classes and that are built using SBT 0.13+ and SCALA 2.11+. We selected projects with more than $1,000$ LOC of production and of test code and using SCALATEST, the most popular testing framework for SCALA. This subset of projects may not be representative of industrial software systems, and therefore replications in this context are desirable.

Finally, our diffusion results might not hold for other test smells. We covered a comprehensive set of 6 test smells, which were reported as the most diffused among JUNIT tests by existing studies [4], [16]. Analysing the diffusion of additional test smells is therefore on our future research agenda.

VIII. CONCLUSION AND FUTURE WORK

Earlier research has demonstrated that test smells can impact test effectiveness, understanding, and maintenance negatively —but has mostly been limited in scope to the combination of JAVA with the JUNIT testing framework. Other language and framework combinations may give developers different options for the design and implementation of their unit tests.

Our results from 164 open-source projects show that test smells are less diffused in SCALA tests that use SCALATEST than in JAVA tests that use JUNIT. We also observe that developers are often unable to perceive and even less to identify test smells —as was already the case for the JAVA and JUNIT combination. Therefore, we argue for more language and framework diversity in empirical research on test smells, as well as research to assess the impact and harmfulness of test smells.

The difference between the languages is even more pronounced when analyzing the GENERAL FIXTURE smell. For this reason, we deem that language designers should provide support for fine-grained definitions of test fixtures and that developers should use more fine-grained means such as "loan fixture methods" and "fixture context objects" for defining and sharing case-specific fixtures instead of using general fixtures.

References

[1] K. Beck, *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[2] B. Beizer, *Software testing techniques*. Dreamtech Press, 2003.

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

[4] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*. IEEE, 2012, pp. 56–65.

[5] F. Palomba and A. Zaidman, "Does refactoring of test smells induce fixing flaky tests?" in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 1–12.

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

[7] M. Tufano, F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, A. De Lucia, and D. Poshyvanyk, "An empirical investigation into the nature of test smells," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 4–15.

[8] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The scala language specification," 2007.

[9] R. Kuhn, B. Hanafee, and J. Allen, *Reactive design patterns*. Manning Publications Company, 2017.

[10] M. Nash and W. Waldron, *Applied Akka Patterns: A Hands-On Guide to Designing Distributed Applications*. " O'Reilly Media, Inc.", 2016.

[11] J. D. Bleser, D. D. Nucci, and C. D. Roover, "Assessing Diffusion and Perception of Test Smells in Scala Projects," 3 2019. [Online]. Available: https://figshare.com/articles/Assessing_Diffusion_and_Perception_of_Test_Smells_in_Scala_Projects/7836332

[12] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "Are test smells really harmful? an empirical study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

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

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

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

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

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

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

[19] F. Palomba, G. Bavota, M. Di Penta, R. Oliveto, D. Poshyvanyk, and A. De Lucia, "Mining version histories for detecting code smells," *IEEE Transactions on Software Engineering*, vol. 41, no. 5, pp. 462–489, May 2015.

[20] F. Palomba, M. Zanoni, F. A. Fontana, A. De Lucia, and R. Oliveto, "Toward a smell-aware bug prediction model," *IEEE Transactions on Software Engineering*, 2017.

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

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

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

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

[25] F. Palomba, G. Bavota, M. Di Penta, F. Fasano, R. Oliveto, and A. de Lucia, "On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation," *Empirical Software Engineering*, vol. 23, pp. 1188–1221, 2018.

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

[27] F. Palomba, A. Panichella, A. Zaidman, R. Oliveto, and A. De Lucia, "The scent of a smell: An extensive comparison between textual and structural smells," *IEEE Transactions on Software Engineering*, 2017.

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

[29] D. I. K. Sjøberg, A. F. Yamashita, B. C. D. Anda, A. Mockus, and T. Dybå, "Quantifying the effect of code smells on maintenance effort," *IEEE Trans. Software Eng.*, vol. 39, no. 8, pp. 1144–1156, 2013.

[30] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad (and whether the smells go away)," *IEEE Transactions on Software Engineering*, vol. 43, no. 11, pp. 1063–1088, 2017.

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

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

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

[34] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800–817, Dec 2007.

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