SoCRATES - Scala Radar for Test Smells

Jonas De Bleser
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
jonas.de.bleser@vub.be

Dario Di Nucci
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
dario.di.nucci@vub.be

Coen De Roover
Software Languages Lab
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

Abstract

Test smells are indications of poorly designed unit tests. Previous studies have demonstrated their negative impact on test understanding and maintenance. Moreover, surveys show that developers are not able to identify test smells, hindering optimal software quality. Automated tools can aid developers to handle these issues and detect test smells in the early stage of software development. However, few tools are publicly available and all of them target JUnit — the most popular testing framework in Java. To overcome these limitations, we propose SoCRATES. This fully automated tool is able to identify six test smells in ScalaTest which is the most prevalent testing framework in Scala. An empirical investigation on 164 Scala projects shows that our tool is able to reach a high precision without sacrificing recall. Moreover, the results show that Scala projects have a lower diffusion than Java projects. We make SoCRATES publicly available as an IntelliJ IDEA plugin, as well as an open-source project in order to facilitate the detection of test smells.

Keywords Scala, Software Quality, Test Smells, Tool

1 Introduction

Test code should be carefully designed by following good programming practices [3, 13]. To this end, van Deursen *et al.* [16] introduced test smells for xUnit, a general purpose framework to test applications. This catalogue takes into account different types of bad design choices made by developers during the implementation of single test cases (*e.g.,* tests checking several methods of the class to be tested), or test fixtures (*e.g.,* tests that only access access a part of a fixture).

Previous studies demonstrated that test smells have a negative impact on software maintenance [2, 6]. Furthermore, developers fail to identify their presence [6, 15]. The majority of these studies concerns the tests written in JUnit which is the most popular testing framework in Java. Despite the introduction and adoption of new programming languages over the years, there is a lack of replication studies for these other languages. Scala [10], for instance, has enjoyed a steady rise in popularity over the past years — and for distributed systems in particular (*e.g.,* [8, 9]). Some of Scala's characteristics have enabled the design of different unit test automation frameworks with sometimes unique features. We believe that these characteristics might impact our understanding of test smells.

Scala'19, July 15–19, 2019, London, United Kindom
2019.

In this paper, we present SoCRATES (**SC**ala **RA**dar for **TE**st **S**mells), a novel tool that automatically detects six test smells within Scala software projects. In order to improve its precision, SoCRATES relies on both syntactic (*i.e.,* abstract syntax trees) and semantic information (*i.e.,* types and symbols). We conducted an evaluation to verify the ability of SoCRATES to detect these test smells where we ran it on a set of 377 test cases and manually compared the results given by the tool with a manually-built oracle. We observed that SoCRATES has a precision of 98.94% and a recall of 89.59% —in line with those achieved by state-of-the-art tools for test smell detection [1, 12].

Afterwards, we employed SoCRATES in an empirical investigation on the diffusion of test smells in Scala source code [5]. The results of this study shows that the majority of Scala projects are affected by at least one code smell, but that their diffusion is low across test classes —with Lazy Test, Eager Test, and Assertion Roulette as the most prevalent ones.

Tool and Source Code. We make our tool publicly available as an IntelliJ IDEA plugin, and publish the corresponding source code in an online appendix [4]. The data used in the experiment can be found in the full paper [5].

Structure of the paper. Section 2 provides a background on test smells in Scala projects, describing an example of a detection rule and the corresponding refactoring. Section 3 presents SoCRATES and how to use it, while Section 4 reports a summary of the previous study where we employed the tool. Finally, Section 5 concludes the paper.

2 Background

This section reports on the definitions of each test smell supported by SoCRATES. We implemented six out eleven test smells defined by van Deursen *et al.* [16], which are the most prevalent in previous work [1, 2, 6, 7, 11, 12, 14, 15], as well as added three variations of General Fixture which are specific to ScalaTest. It is worth considering that in a previous contribution [5] we transposed the original definitions to Scala and ScalaTest, defined a static method for their detection, and proposed the refactoring required to eliminate them. We considered test cases implemented in ScalaTest, because in our previous investigation we found that this is the most prevalent testing framework for Scala. Nevertheless, the tool is open-source and developers are kindly invited to submit pull requests with the aim of implementing additional detection methods.

Assertion Roulette (AR). A test case that contains more than one assertion of which at least one does not provide a reason for assertion failure. This test smell encumbers identifying which assertion failed and the reason why.

Eager Test (ET). A test case that checks or uses more than one method of the class under test. It is left to interpretation which method calls count towards the maximum, but we consider all methods invoked on the class under test. A solid principle is that every test case should test only one method such that the test only fails when the single method fails, and not because another irrelevant method fails.

General Fixture (GF). 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 four different means for defining and sharing fixtures. Yet, the detection method and refactoring for these variations differ. This smell impacts the maintainability, since remove fields from the fixture might affect multiple unrelated tests.

Lazy Test (LT). 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 [16] 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. The drawback of having this smell implies that you have to modify multiple tests, instead of a single one.

Mystery Guest (MG). 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 behaviour of the test.

Sensitive Equality (SE). 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()`. This makes the test vulnerable to small, and irrelevant details (*e.g.,* spaces).

Along with the definition of these smells, in our previous contribution [5] we reported: (i) an example instance, (ii) a static detection method based on the rules introduced by Bavota *et al.* [1, 2], and (iii) a refactoring to eliminate the smell and its negative impact. Only for General Fixture we provided four detection methods based on the different implementations of test fixture in ScalaTest: Global Fixture, Loan Fixture, Fixture Context, and With Fixture.

For illustration purposes, we report the detection method and the refactoring for Fixture Context.

Fixture Context: Detection Method. "Fixture context" objects are instances, such as the ones instantiated on lines 14 and lines 20 of Listing 1, of an anonymous class that mixes in at least one trait such as `RecipeFixture` that provides *and* initialises fields for the fixture. The body of the anonymous class itself corresponds to the test case, such as the `assert` expressions on lines 16 and 21. 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 General Fixture test smell. The detection rule 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).

```
1  class RecipeTestSuiteFCO extends FlatSpec {
2
3    trait RecipeFixture {
4      val ingredients1 =
5        List(Ingredient("Cookie", 100), Ingredient("Milk", 200))
6      val ingredients2 =
7        List(Ingredient("Eggs", 100), Ingredient("Bacon", 200))
8      val cookiesAndMilk =
9        Recipe("Cookies and Milk", ingredients1)
10     val baconAndEggs = Recipe("Eggs", ingredients2)
11   }
12
13   "The recipe" should "have two ingredients (Eggs, Bacon)" in
14     new RecipeFixture {
15       val r = baconAndEggs.names.equals(List("Eggs", "Bacon"))
16       assert(r == true, "...")
17   }
18
19   "The recipe" should "have two ingredients" in
20     new RecipeFixture {
21       assert(cookiesAndMilk.ingredients.size == 2, "...")
22   }
23 }
```

Listing 1. General Fixture Example.

Fixture Context: Refactoring. The refactoring shown below in Listing 2 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.

```
class RecipeTestSuiteFCOR extends FlatSpec {

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

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

Listing 2. General Fixture Refactoring.

3 SoCRATES

In the following sections we explain how SoCRATES works internally, as well as its usage, input and output.

3.1 Internal Working of SoCRATES

SoCRATES is implemented in Scala, and extensively uses the library scala-meta[1] (and SemanticDB). The tool takes a project as input and produces an overview of the detected test smells according to the test smell detection rules. This process consists of several steps:

#1: Code Compilation. To precisely detect test smells, SoCRATES does not only rely on syntactic information (*i.e.,* abstract syntax trees), but also on semantic one such as types and symbols. Therefore, every project must be compiled with a compiler plugin[2] that is copied in the source code of the target project in background. To generate the SemanticDB, the `sbt semanticdb` command is automatically executed afterwards.

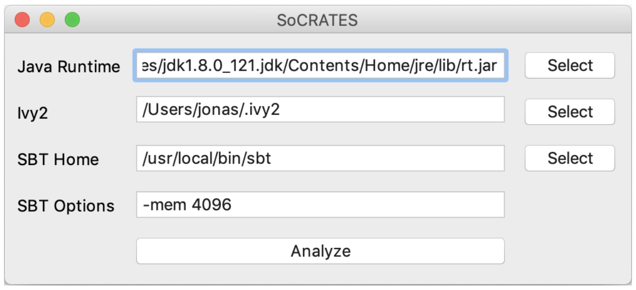
#2: Extraction of Syntactic and Semantic Information. After this step SoCRATES builds the class hierarchy by collecting information of all classes (*i.e.,* their type, their fully qualified name and their parents) and link them together by traversing the inheritance chain. This hierarchy will be used in a later step to determine which test classes use a ScalaTest specification.

#3: Identification of Test Classes. Each project consists of production and test classes. This classification is done based on the typical structure of Scalaprojects that are build with the Scala Build Tool[3] (SBT): all Scala files in `src/main/scala` are production classes, while those in `src/test/scala` are test classes. Subsequently, we filter the tests classes to only keep those that really represent a test class, and not simply an auxiliary class. To this aim, we rely on the computed class hierarchy to check whether the test class inherits from a well-known test specification class. The tool supports most of the specifications from ScalaTest: `FlatSpec`, `FunSuite`, `WordSpec`, `FunSpec`, `FreeSpec`, `FeatureSpec`, `PropSpec`, and `RefSpec`. For every specification of ScalaTest, we also check some variants: `{s}Like`, `Async{s}` and `Async{s}Like` where `{s}` is a specification. For example, we will also check for the following four specifications for `FunSpec`: `FunSpec`, `FunSpecLike`, `AsyncFunSpec` and `AsyncFunSpecLike`.

#4: Linking Test Classes to Production Classes. Each test class targets a specific production class. To identify this class, we adopted a widely-used practise based on code convention. Given the name of the a test class, SoCRATES removes one of the following suffixes from it: `Test`, `Tests`, `TC`, `TestCase`, `Spec`, `Specification`, `Suite`, `Prop`. Next, it checks whether such a class with the same name and package exists. For example, given the `ShoppingCartTests` test class in the package `be.vub.soft`, the suffix `Tests` is found and removed from the original name. This results in a production class that should exist in the package `be.vub.soft` and have the name `ShoppingCart`. In this example, we link a production class to a test class if and only if there exists a production class with the same name and which resides in the same package. Test class instances are further analysed to generate the list of test cases.

#5: Test Smell Detection. Finally, test smells are detected for each test class and and test cases. Indeed, while some test smells are specific to test classes, most of them are specific to test cases. The test smell detection methods are implemented by extending the classes `TestClassCodeSmell` and `TestCaseCodeSmell` respectively.

[1]https://scalameta.org
[2]https://gist.github.com/olafurpg/a74404dfee6b3da03892af17357074d9
[3]https://github.com/sbt/sbt



Java Runtime: es/jdk1.8.0_121.jdk/Contents/Home/jre/lib/rt.jar
Ivy2: /Users/jonas/.ivy2
SBT Home: /usr/local/bin/sbt
SBT Options: -mem 4096

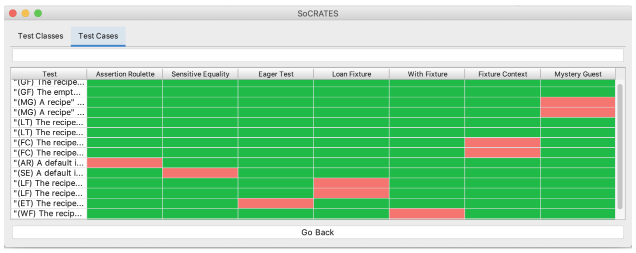
Figure 1. SoCRATES's Option View

3.2 How to Use SoCRATES

SoCRATES is available as an IntelliJ IDEA plugin that can be downloaded from the online appendix [4]. The plugin must be installed before it can be used within a project. Figure 1 shows the options that can be used to configure the process. The first option specifies the location of the `rt.jar`. This is needed to build the SemanticDB of the Scala standard library. The second option is the location of the `ivy2` cache. This folder contains all of external dependencies which are required to successfully compile the project. These .jar files are required to build the SemanticDB. Multiple folders can be separated by colons (*e.g.,* `path1:path2`). The third option specifies the location of the SBT binary. This is required as SoCRATES will execute a SBT task that activates a compiler plugin in the background. The final option enables the developer to specify custom SBT options. By default, this includes an increased memory allocation because building the SemanticDB for a project can require a significant amount of memory.

3.3 SoCRATES Output

Figure 2 shows the output produced by SoCRATES on a sample project. It consists of two tables: the first one reports on the test smells Global Fixture and Eager Test related to test classes; while the second one reports on the test smells related to test cases. The results can be sorted by column and filtered to quickly investigate whether a given test class or test cases exhibits test smells.



Test	Assertion Roulette	Sensitive Equality	Eager Test	Loan Fixture	With Fixture	Fixture Context	Mystery Guest
"(GF) The recipe...	🟩	🟩	🟩	🟩	🟩	🟩	🟩
"(GF) The empt...	🟩	🟩	🟩	🟩	🟩	🟩	🟩
"(MG) A recipe" ...	🟩	🟩	🟩	🟩	🟩	🟩	🟥
"(MG) A recipe" ...	🟩	🟩	🟩	🟩	🟩	🟩	🟥
"(LT) The recipe...	🟩	🟩	🟩	🟩	🟩	🟩	🟩
"(LT) The recipe...	🟩	🟩	🟩	🟩	🟩	🟩	🟩
"(FC) The recipe...	🟩	🟩	🟩	🟩	🟩	🟥	🟩
"(FC) The recipe...	🟩	🟩	🟩	🟩	🟩	🟥	🟩
"(AR) A default i...	🟥	🟩	🟩	🟩	🟩	🟩	🟩
"(SE) A default i...	🟩	🟥	🟩	🟩	🟩	🟩	🟩
"(LF) The recipe...	🟩	🟩	🟩	🟥	🟩	🟩	🟩
"(LF) The recipe...	🟩	🟩	🟩	🟥	🟩	🟩	🟩
"(ET) The recipe...	🟩	🟩	🟥	🟩	🟩	🟩	🟩
"(WF) The recip...	🟩	🟩	🟩	🟩	🟥	🟩	🟩

Figure 2. SoCRATES' Results View

4 Evaluation

In this section we summarise the empirical validation of SoCRATES and the empirical study on the test smells diffusion across Scala projects that we conducted afterwards.

Indeed, our first goal was to assess the ability of the tool in detecting test smells in Scala projects. To this end, we manually validated a subset of 377 test cases by comparing the outcome of the tool and a manual inspection. These test cases were a statistically significant sample from a dataset composed of 164 projects. These projects (i) were created on Github between January 2010 and July 2018, (ii) have test classes, (iii) use SBT for build automation (required to obtain semantic information), and (iv) compile without any error. Table 1 shows the dataset characteristics.

Table 1. Dataset Characteristics.

	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

We observed that SoCRATES has a precision of 98.94% and a recall of 89.59% which is in line with those achieved by state-of-the-art tools for test smell detection [1, 12]. The detailed results are reported in Table 2.

We also conducted an empirical study with the *goal* of analysing test smell diffusion in open-source Scala systems 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.

Table 2. Percentage of Projects, Test Classes, and Test Cases Exhibiting the Smells along with Precision and Recall of SoCRATES 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%

In summary, the results show that:

- 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 [1, 2].
- Lazy Test (62.19%), Eager Test (51.82%), and Assertion Roulette (44.51%) are the three most prevalent test smells across Scala projects, while General Fixture (27.41%), Mystery Guest (15.85%), and Sensitive Equality (13.41%) are the least prevalent.

5 Conclusion

In this paper, we present SoCRATES — a tool to detect six test smells in Scala project that use the popular testing framework ScalaTest. Our tool leverages detection rules that are implemented by means of both syntactic and semantic information. The former is obtained by traversing source code with scala-meta, while the latter is obtained by building SemanticDB and querying types and symbols from it.

We evaluate the detection performances of SoCRATES with respect to a manually validated oracle and observed that it is able to achieve high precision and recall. Therefore, we employ the tool in an empirical study [5] to investigate the diffusion of test smells across 164 Scala projects. Our results show that test smells are less diffused in Scala when used in combination with ScalaTest than in Java tests that use JUnit.

We make SoCRATES publicly available for researchers and developers, both as an open-source project and as an IntelliJ IDEA plugin. We encourage users to implement their own detection rules as SoCRATES is built to be extendable from the ground up.

References

[1] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David Binkley. 2012. 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, 56–65.

[2] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and Dave Binkley. 2015. Are test smells really harmful? An empirical

study. *Empirical Software Engineering* 20, 4 (2015), 1052–1094. DOI: http://dx.doi.org/10.1007/s10664-014-9313-0

[3] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[4] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Appendix: SoCRATES - Scala Radar for Test Smells. (4 2019). DOI: http://dx.doi.org/10.6084/m9.figshare.7971014.v2

[5] Jonas De Bleser, Dario Di Nucci, and Coen De Roover. 2019. Assessing Diffusion and Perception of Test Smells in Scala Projects. In *Proceedings of the 16th International Conference on Mining Software Repositories.*

[6] M. Greiler, Arie van Deursen, and M.-A. Storey. 2013. Automated Detection of Test Fixture Strategies and Smells. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST).* 322–331. DOI: http://dx.doi.org/10.1109/ICST.2013.45

[7] Michaela Greiler, Andy Zaidman, Arie van Deursen, and M.-A. Storey. 2013. Strategies for Avoiding Text Fixture Smells During Software Evolution. In *Proceedings of the 10th Working Conference on Mining Software Repositories (MSR).* IEEE, 387–396.

[8] Roland Kuhn, Brian Hanafee, and Jamie Allen. 2017. *Reactive design patterns.* Manning Publications Company.

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

[10] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. 2007. The Scala language specification. (2007).

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

[12] Fabio Palomba, Dario Di Nucci, Annibale Panichella, Rocco Oliveto, and Andrea De Lucia. 2016. 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, 5–14.

[13] A. Schneider. 2000. JUnit best practices *(Java World).* DOI: http://dx.doi.org/javaworld/jw-12-2000/jw-1221-junit.html

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

[15] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. 2016. An Empirical Investigation into the Nature of Test Smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016).* ACM, New York, NY, USA, 4–15. DOI: http://dx.doi.org/10.1145/2970276.2970340

[16] Arie van Deursen, Leon Moonen, Alex Bergh, and Gerard Kok. 2001. Refactoring Test Code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering (XP).* 92–95.