Just-In-Time Test Smell Detection and Refactoring: The DARTS Project

Stefano Lambiase
SeSa Lab - University of Salerno
Fisciano (SA), Italy
s.lambiase7@studenti.unisa.it

Andrea Cupito
SeSa Lab - University of Salerno
Fisciano (SA), Italy
a.cupito@studenti.unisa.it

Fabiano Pecorelli
SeSa Lab - University of Salerno
Fisciano (SA), Italy
fpecorelli@unisa.it

Andrea De Lucia
SeSa Lab - University of Salerno
Fisciano (SA), Italy
adelucia@unisa.it

Fabio Palomba
SeSa Lab - University of Salerno
Fisciano (SA), Italy
fpalomba@unisa.it

ABSTRACT

Test smells represent sub-optimal design or implementation solutions applied when developing test cases. Previous research has shown that these smells may decrease both maintainability and effectiveness of tests and, as such, researchers have been devising methods to automatically detect them. Nevertheless, there is still a lack of tools that developers can use within their integrated development environment to identify test smells and refactor them. In this paper, we present DARTS (Detection And Refactoring of Test Smells), an Intellij plug-in which (1) implements a state-of-the-art detection mechanism to detect instances of three test smell types, i.e., General Fixture, Eager Test, and Lack of Cohesion of Test Methods, at commit-level and (2) enables their automated refactoring through the integrated APIs provided by Intellij.

Video. https://youtu.be/sd3V2J7k8Zs
Source Code. https://github.com/StefanoLambiase/DARTS

CCS CONCEPTS

• **Software and its engineering** → **Software maintenance tools**.

KEYWORDS

Test Smells, Refactoring, Software Testing.

ACM Reference Format:
Stefano Lambiase, Andrea Cupito, Fabiano Pecorelli, Andrea De Lucia, and Fabio Palomba. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3387904.3389296

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389296

1 INTRODUCTION

Regression testing is the activity that allows developers to run test cases as soon as a new change is committed onto a repository with the aim of finding possible faults introduced by the modification [2]. To this aim, with the help of testing framework (e.g., JUnit[1]) developers write tests to enable regression analysis. However, while developing them, programmers sometimes introduce sub-optimal implementation solutions that have the potential impact of reducing test code understandability, maintainability, and effectiveness: these are called *test smells* [6]. Previous research has investigated these smells under different angles, highlighting that tests affected by them have reduced fault detection capabilities [9], other than requiring more effort to be maintained and/or comprehended [1, 4].

In the recent past, researchers have been studying methods to support developers with the identification of test smells. In particular, earlier approaches were defined by van Rompaey et al. [10] and Greiler et al. [5], who defined detection strategies based on the computation of code metrics. For example, van Rompaey et al. [10] proposed to identify instances of *Eager Test*, i.e., the smell that arises when a non-cohesive test exercises multiple methods of the production class [6], by looking at the number of method calls made by a test: if this is higher than a predefined threshold, then an *Eager Test* is detected. More recently, Palomba et al. [8] defined a novel technique, named Taste, which implements an information retrieval mechanism to identify three types of test smells such as *Eager Test*, *General Fixture*, and *Lack of Cohesion of Test Methods*. The empirical assessment of Taste showed that it can overcome the performance of code metrics-based detectors, hence being potentially more useful in practice.

Despite the effort spent by researchers so far, none of the proposed detection approaches has been implemented into a tool usable by developers. Furthermore, to the best of our knowledge, there is no tool able to support developers with the automatic refactoring of test smells. As such, the practical relevance of the research made in the field is still limited.

To overcome these limitations, we develop and propose Darts (which is the acronym of **D**etection **A**nd **R**efactoring of **T**est **S**mells), an Intellij plugin that (1) enables the identification of the three test smell types originally supported by Taste [8] and (2) supports their removal through automatic refactoring. We implement

[1] https://junit.org/

DARTS so that it can work at commit-level, hence implementing a *just-in-time* philosophy that allows developers to discover and refactor test code as soon as a new commit is performed. We make the plugin publicly available and open-source on GITHUB, in an effort of encouraging the research community to further extend our tool with additional test smells and refactoring options.

2 TEST SMELL DETECTION AND REFACTORING IN DARTS

This section reports details about the test smells involved in our tool. For each of them we provide a definition, then we report details about the automatic detection, following the detection rules proposed by Palomba et al. [8], and finally we present the automatic refactoring performed by DARTS.

2.1 General Fixture

Definition There is a *General Fixture* when the fixture (initialization of setup variables) of a test suite is too general and the test methods in the suite only use a part of it [6].

Identification The detection rule by Palomba et al. [8] is based on the calculation of the number of disjoint pairs of methods in a test class, where two methods are defined *disjoint* if they have a cosine similarity = 0 and they use parts of the setup method. The number of disjoint pairs of methods are then used to compute the percentage of disjoint pairs as follows:

$$P_{GF}(T) = \frac{|Disjoint\_Pairs|}{|\{(t_i, t_j) : t_i, t_j \in T\}|} \quad (1)$$

Finally, for each test class T, if $P_{GF}(T)$ is greater than 0.6, then T is considered as a General Fixture.

Refactoring Once identified one (or more) class affected by General Fixture test smell, DARTS provides an automatic refactoring proposal based on the following implementation:

(1) First of all, DARTS identifies the method in the test suite that causes General Fixture. For the sake of comprehensibility let's call it m_{GF};
(2) Then, DARTS collects the list of all the instance variables used by such method;
(3) DARTS performs an Extract Method Refactoring in order to split the set up in two methods. The first will contain only the instance variables used by the m_{GF}; the second one will contain all the remaining instance variables. In so doing, DARTS exploits the data flow analysis engine of the INTELLIJ APIs[2] in order to identify local variables and parameters that are used inside a method, but that are declared outside of it.
(4) Finally, DARTS performs an Extract Class Refactoring creating a new class that will contain (i) the instance variables used by m_{GF}, (ii) the set up method for their initialization, and (iii) the method m_{GF}.

2.2 Eager Test

Definition *Eager Test* occurs when a test method exercises more than one method of the production class under test [6].

[2]https://www.jetbrains.com/help/idea/analyzing-data-flow.html

Identification Given a test method t, TASTE [8] computes the probability that t is affected by Eager Test as follows:

$$P_{ET}(t) = 1 - TestMethodCohesion(t) \quad (2)$$

where TestMethodCohesion(t) is the average textual similarity between all the pairs of methods called by t. Also in this case, a 0.6 threshold is used to establish if the method is affected or not by the smell.

Refactoring Given a method affected by Eager Test, DARTS proposes an automatic Extract Method Refactoring operation. In particular, it is able to identify the portions of code that cause Eager Test and move them to a new method.

2.3 Lack of Cohesion of Test Methods

Definition There is a Lack of Cohesion of Test Methods when there is low cohesion between the test methods in a test suite.

Identification Given a test class, we rely on the TASTE detection strategy [8] in which a test class (t) is considered to be affected by Lack of Cohesion of Test Methods if $P_{LCTM}(t) > 0.6$.

$$P_{LCTM}(T) = 1 - TestClassCohesion(T) \quad (3)$$

where TestClassCohesion(T) is the average textual similarity between all the methods in the class T.

Refactoring Given a class affected by Lack of Cohesion of Test Methods, DARTS proposes an Extract Class Refactoring in which the methods with lower cohesion with the test class are moved to a new class. Note that if such methods use part of the fixture, DARTS also performs an Extract Method Refactoring in order to split the set up method and move it to the new class.

3 DARTS: A USE CASE SCENARIO

In this section we report a use case scenario of how DARTS can be used by developers—the companion video shows in more detail the scenario described herein.

Let suppose that when maintaining the system she is working on, a developer modifies the production classes `Example1` and `Example2` and, accordingly, makes a consistent change to the corresponding test suites, called `Example1Test` and `Example2Test`, by introducing test smell instances. When committing her changes onto the repository, DARTS automatically warns the developer about the existence of some test smell instances in the tests she has modified. At this point, the developer can either decide to ignore the warning, hence continuing with the commit, or review the test files included in the newly committed changes. If the second option is selected, DARTS shows the window depicted in Figure 1. In particular, the plugin provides the developer with three main tabs, one for each supported test smell type. For each of them, DARTS shows three pieces of information. Under the *'Classes'* tab (panel ① in the figure), it indicates which test suite contains at least one instance of test smells, while under the *'Methods'* tab (panel ②) the developer can see the list of test methods involved in a smell. More importantly, the right-hand side of the window allows the

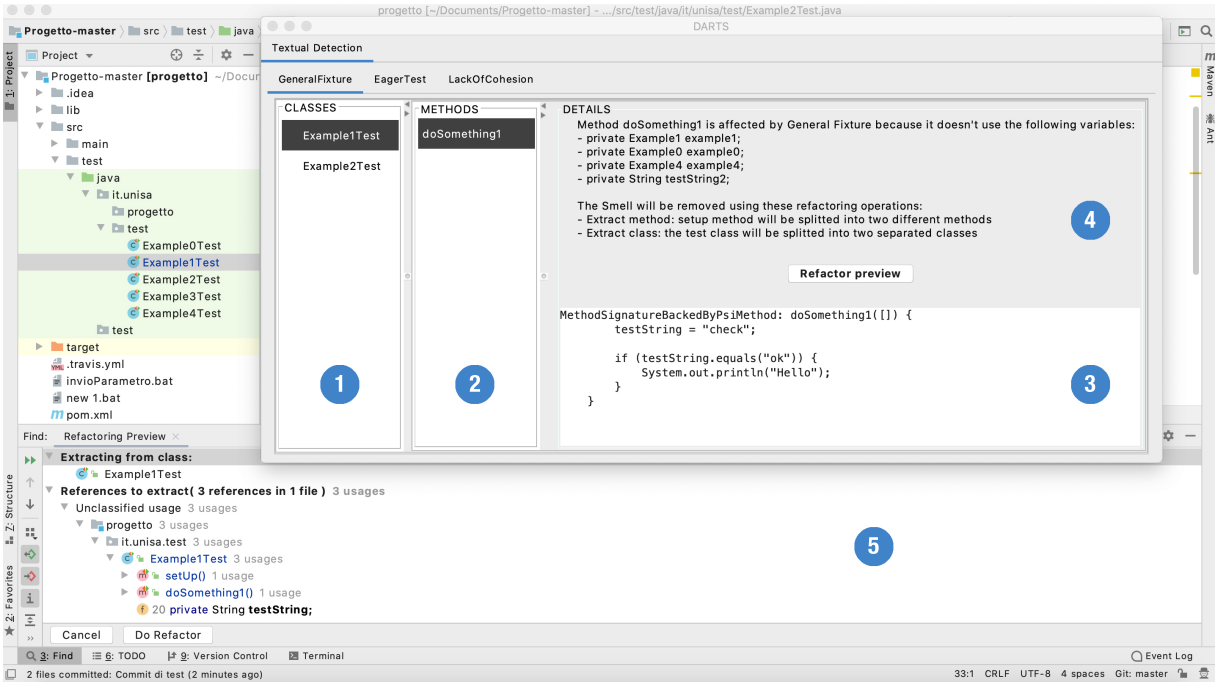

Figure 1: DARTS - Main window showing the results of the test smell detection process.

developer to directly analyze the smelly test method (panel ③) as well as gather tips on how to refactor it (panel ④). The template used to give tips is automatically adapted based on the type of test smell affecting the test under review. On the one hand, the explanation of the smell comes directly from the analysis of how it is detected: in the example shown in Figure 1, a *General Fixture* is identified because the test `ExampleTest1.doSomething1` only partially uses the fixture of the class, thus making the test refactorable. On the other hand, the text used to provide refactoring suggestions is static and based on the specific operation associated to the smell: in the figure, the developer receives the list of possible actions that could be performed to refactor the code, i.e., an *Extract Method* and *Extract Class* refactoring.

As a final step, the developer can decide to click the *'Refactoring Preview'* button: it will open a new dialog (panel ⑤) where she can have indications of how the refactoring will be performed. In this specific use case scenario, the developer will see all the elements (e.g., methods, variables) involved in the two suggested refactoring operations (*Extract Method* and *Extract Class*). Afterwards, the developer can decide to apply a refactoring and, in this case, the tool will automatically modify the source code accordingly.

4 ARCHITECTURE OF THE TOOL

Figure 2 overviews the architecture of DARTS, which has been developed through the INTELLIJ and JAVA 8 APIs. It consists of two main layers, which are described in the following:

Interface Layer. The interactions with the user as well as the logic pertaining to the creation of the contents to be displayed are managed in this layer. Two subsystems contribute to this component. First, the `extensions` package contains the classes required to make the plugin working at commit-level: specifically, the package implements the feature that allows DARTS to keep listening the activities made by a developer and interact with her when she performs a commit onto a remote repository. Finally, the `toolWindowsConstruction` package is the one responsible for the definition of the graphical user interface and the UI components that allow the user to interact with the detection analysis and refactoring functionalities implemented by the tool.

Application Layer. The whole logic of the plugin, including the detection and refactoring mechanisms, is placed in this layer. Here there are two main subsystems which we named `testSmellDetection` and `refactoring`. The former contains the test smell detection rules, one for each smell considered. In this regard, it is important to note that we implemented a *Strategy* design pattern [3] that defines a common interface, named `ITestSmellDetection`, that eases the extensibility of DARTS, allows other researchers to implement new detection rules without compromising the existing ones, and enables the definition of a strategy to dynamically change the detection mechanism based on some condition. The second subsystem concerns with the refactoring operations implemented by the tool. Classes in this package are called once the detection phase is completed with the aim of manipulating the smelly test code and modifying the source code according to the refactoring that the developer wants to perform. From a practical perspective, the subsystem heavily relies on PSI[3] (that stands for **P**rogram **S**tructure **I**nterface), the INTELLIJ library for code manipulation: it allows to navigate

[3]https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/navigating_psi.html

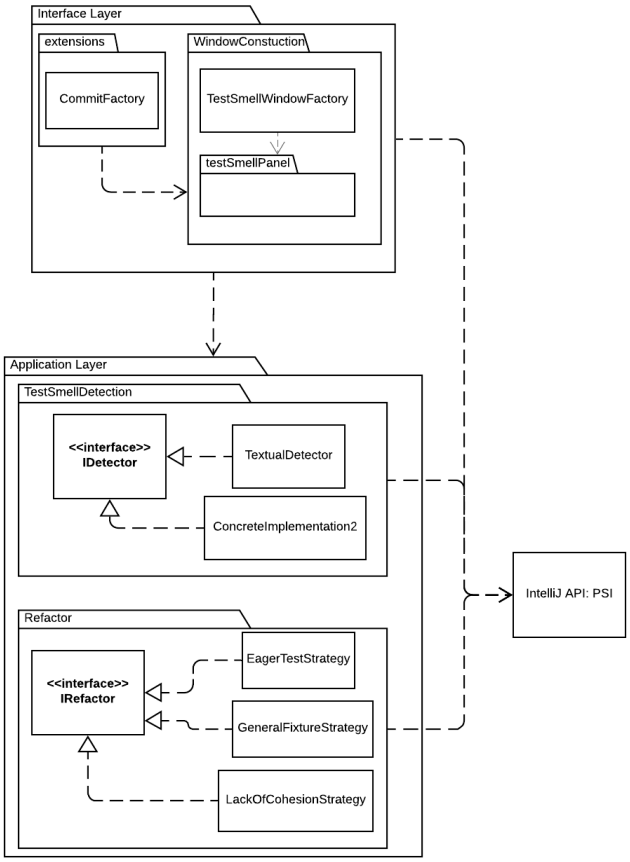

Figure 2: DARTS architecture

the source code elements and modify them implementing the refactoring actions associated to the considered test smells. It is worth noting that, as done in the previous case, we implemented a *Strategy* design pattern to define an interface called `IRefactor`, that could enable the definition of different refactoring strategies than those applied in the context of this tool demo.

5 EVALUATION OF THE TOOL

The performance of Darts strongly depends on the identification mechanism it implements. As explained in Section 2, the proposed tool builds on top of Taste, a textual-based detector we previously designed and evaluated [8]. More specifically, the test smell detector has been ran against a manually-validated oracle reporting a total amount of 494 actual test smell instances pertaining to 12 open-source software projects coming from the Apache Software Foundation—21 of these instances related to *General Fixture*, 268 to *Eager Test*, and 205 to *Lack of Cohesion of Test Methods*.[4]

On the one hand, Taste was evaluated in terms of detection capabilities through the computation of precision, recall, and F-Measure. On the other hand, it has been empirically compared to code metrics-based detectors such as those by van Rompaey et al. [10] and Greiler et al. [5]. From this two-step assessment, we first discovered that Taste can identify *General Fixture*, *Eager Test*, and *Lack of Cohesion of Test Methods* with an overall F- Measure of 67%, 76%, and 62%, respectively. Furthermore, the detector improves upon the metrics-based techniques by up to 44%, 29%, and 9%, respectively, for the three smell types. This practically indicated that Taste could better support the developer's activities when compared to the baselines. Nonetheless, the performance could still be improved, for instance by combining different sources of information. This is the main reason why we decided to release Darts as an open-source product: researchers can build upon our tool to improve its performance and directly see their results implemented in a plugin usable by developers.

As for the refactoring mechanisms available in our tool, it is first worth discussing the fact that we do not rely on any refactoring recommender to suggest how to refactor test smells. This was a conscious decision that came from two observations. In the first place, to the best of our knowledge there is no refactoring recommender available in literature and, as such, this is something that goes beyond the scope of this tool demo but that can potentially be integrated in our tool. In the second place, most of the refactoring operations implemented by Darts follow the removal steps suggested in the test smell and refactoring catalog defined by Meszaros [6], which can be easily implemented through the use of the IntelliJ APIs. For these reasons, we deem this initial refactoring implementation as satisfactory and helpful for providing developers with a useful tool to improve test code quality.

As a final note, Darts directly exploits the IntelliJ APIs when applying refactoring operations. This first ensures that the resulting source code is compilable and error-free. When it comes to behavior preservation, Darts implements refactoring actions that are supposed to only improve quality aspects of test code without altering the way it exercises production classes. Nevertheless, we implemented the *'Undo'* functionality to allow developers to restore previous versions of test code in case they realize some change in the expected behavior of the tests.

6 DEMO REMARKS

In this paper, we presented Darts, an IntelliJ plugin that enables the detection and refactoring of three test smell types such as *Eager Test*, *General Fixture*, and *Lack of Cohesion of Test Methods* at commit-level. We made the tool publicly available and open-source with the aim of encouraging the research community in further improving our tool with additional test smells, detectors, and/or refactoring recommendations.

Our future research agenda includes (1) the integration of other detectors, e.g., the code metrics-based approaches by van Rompaey et al. [10] and Greiler et al. [5] and (2) the evaluation of how Darts is used in practice: to this aim, we plan to release the tool in the IntelliJ plugin store and make it downloadable by developers.

ACKNOWLEDGMENTS

Palomba gratefully acknowledges the support of the Swiss National Science Foundation through the SNF Project No. PZ00P2_186090.

REFERENCES

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

[4] The dataset is also publicly available in the Landfill platform [7].

Software Engineering 20, 4 (2015), 1052–1094.

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

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1993. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 406–431.

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

[5] Michaela Greiler, Arie Van Deursen, and Margaret-Anne Storey. 2013. Automated detection of test fixture strategies and smells. In *Software Testing, Verification and Validation (ICST)*. 322–331.

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

[7] Fabio Palomba, Dario Di Nucci, Michele Tufano, Gabriele Bavota, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2015. Landfill: An open dataset of code smells with public evaluation. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 482–485.

[8] Fabio Palomba, Andy Zaidman, and Andrea De Lucia. 2018. Automatic test smell detection using information retrieval techniques. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 311–322.

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

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