DARTS 2.0

Iuliano Gerardo, La Monica Tiziano

University of Salerno

July 2022

Abstract

Our work is based on the detection of test smell. The aim of the project is to improve the DARTS plug-in by implementing new detectors that are able to detect the presence of one or more test smells. What we achieved is a new release of the plug-in that offers new functionality.

1. Context of the project

DARTS (Detection And Refactoring of Test Smells) is an Intellij plug-in which 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. The plugin allows you to perform automatic refactoring only at the end of the analysis, after having produced the data necessary for the identification and elimination of the Test Smell.

2. Problem Description

Unit test code, just like regular/production source code, is subject to bad programming practices, known also as anti-patterns, defects and smells. Smells, being symptoms of bad design or implementation decisions, has been proven to be responsible for decreasing the quality of software systems from various aspects, such as making it harder to understand, more complex to maintain, and more prone to errors and bugs.

Test smells are defined as bad programming practices in unit test code (such as how test cases are organized, implemented and interact with each other) that indicate potential design problems in the test source code.

3. Solution

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.

Its heart is a series of small behavior preserving transformations. Each transformation (called a "refactoring") does little, but a sequence of these transformations can produce a significant restructuring. Since each refactoring is small, it's less likely to go wrong. The system is kept fully working after each refactoring, reducing the chances that a system can get seriously broken during the restructuring.

First, however, we need to identify the test smells.

4. Goal of the project

The goal of project is improve DARTS by adding new test smell detectors such us Conditional Test Logic, Constructor Initialization, Duplicate Assert, Exception Handling, Ignored Test and Magic Number Test. The change consists in implementing new detectors for tests smell. DARTS implements two types of detection, structural and textual. Researchers have devised tools and techniques to detect code smells in software systems. Most of them are based on the analysis of the properties extractable from the source code (e.g., method calls) by means of a combination of structural metrics, while in recent years the use of alternative sources of information (i.e., historical and textual analysis) have been explored, together with methodologies based on machine learning and search-based algorithms. The textual detection of code smells is able to identify code smells using a three step process, i.e., (i) textual content extraction, (ii) application of IR normalization process, and (iii) application of specific heuristics in order to detect code smells related to promiscuous responsibilities. The detection is going to base on structural information.

5. Test smells and structural rules

To detect the presence of a test smell we need to understand what caused it. Each test smell has one or more characteristics that allow us to be able to identify them. Given the nature of the smells to be identified, analyzing the code from a structural point of view is sufficient to identify the presence of one of them. Some types of test smells are very subjective or depend on the characteristics of the software. Large software can have a tolerance towards a smell different than a smaller software. The presence can be determined by the tuning of some thresholds set by the programmer or by the tester.

5.1. Conditional Test Logic

Test methods need to be simple and execute all statements in the production method. Conditions within the test method will alter the behavior of the test and its expected output, and would lead to situations where the test fails to detect defects in the production method since test statements were not executed as a condition was not met. Furthermore, conditional code within a test method negatively impacts the ease of comprehension by developers.

- Detection: A test method that contains one or more control statements (i.e if, switch, conditional expression, for, foreach and while statement). The detection makes use of a threshold from 0 to 5.

5.2. Constructor Initialization

Ideally, the test suite should not have a constructor. Initialization of fields should be in the setUp() method. Developers who are unaware of the purpose of setUp() method would give rise to this smell by defining a constructor for the test suite.

- Detection: A test class that contains a constructor declaration.

5.3. Duplicate Assert

This smell occurs when a test method tests for the same condition multiple times within the same test method. If the test method needs to test the same condition using different values, a new test method should be utilized; the name of the test method should be an indication of the test being performed. Possible situations that would give rise to this smell include: (1) developers grouping multiple conditions to test a single method; (2) developers performing debugging activities; and (3) an accidental copy-paste of code.

- Detection: A test method that contains more than one assertion statement with the same parameters.

5.4. Exception Handling

This smell occurs when a test method explicitly a passing or failing of a test method is dependent on the production method throwing an exception. Developers should utilize JUnit's exception handling to automatically pass/fail the test instead of writing custom exception handling code or throwing an exception.

- Detection: A test method that contains either a throw statement or a catch clause. The detection makes use of a threshold from 0 to 5.

5.5. Ignored Test

JUnit 4 provides developers with the ability to suppress test methods from running. However, these ignored test methods result in overhead since they add unnecessary overhead with regards to compilation time, and increases code complexity and comprehension.

- Detection: A test method or class that contains the @Ignore annotation.

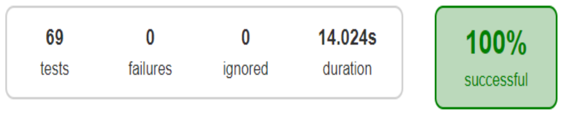
5.6. Magic Number Test

Occurs when assert statements in a test method contain numeric literals (i.e., magic numbers) as parameters. Magic numbers do not indicate the meaning/purpose of the number. Hence, they should be replaced with constants or variables, thereby providing a descriptive name for the input.

- Detection: An assertion method that contains a numeric literal as an argument.

6. Testing

Testing was done using the category partition. The parameters for testing are the project and the threshold. For the project was identified the category *"number of instances"*, while for the threshold the *"values in range"* and the *"values out of range"*. The choices for the number of instances are: 0 instance, 1 instance and 2 or more instance. The choices for values in range are: 0, 2, and 5. While for out of range are: -1 and 6. Furthermore, regression testing was executed to verify the correct functioning of the plugin after the changes.



69	0	0	14.024s	100%
tests	failures	ignored	duration	successful

Figure 1: *Test Result*

7. Test Result

69 tests were obtained, 42 derived from the category partition and the remainder were obtained through a white-box approach and the use of CFG.

References

[1] [Software Unit Test Smell] Web Site: https://testsmells.org

[2] [Refactoring.com] Web Site: https://refactoring.com

[3] [Code Smell: Relevance of the Problem and Novel Detection Techniques] Research Thesis: http://elea.unisa.it/handle/10556/2566