JTDog: a Gradle Plugin for Dynamic Test Smell Detection

Masayuki Taniguchi, Shinsuke Matsumoto, Shinji Kusumoto
Graduate School of Information Science and Technology
Osaka University
Osaka, Japan
{m-tanigt, shinsuke, kusumoto}@ist.osaka-u.ac.jp

***Abstract*—The concept of the test smell represents potential problems with the readability and maintainability of the test code. Common test smells focus on static aspects of the source code, such as code length and complexity. These are easy to detect and do not cause problems in terms of test execution. On the other hand, dynamic smells, which are based on test runtime behavior, lead to misunderstanding of the test results. For example, rotten green tests give developers the false impression that the test was passed without any problems, even though the test was poorly executed. Therefore, we should detect dynamic smells and take countermeasures as early as possible through the development. In this paper, we introduce JTDog, a Gradle plugin for dynamic smell detection. JTDog has high portability due to its integration into the build tool. We applied JTDog to 150 projects on GitHub and confirmed that the JTDog plugin has high portability. In addition, JTDog detected 958 dynamic smells in 55 projects. JTDog is available at https://github.com/kusumotolab/JTDog, and the demo video is available at https://youtu.be/t374HYMCavI.**

***Index Terms*—software testing, test smell, Gradle plugin, build tool, dynamic analysis**

I. Introduction

Test smells represent potential problems with the readability and maintainability of the test code [1]. The presence of test smells can lead to poor test quality, and it is desirable to detect test smells early and eliminate them through code improvements, such as refactoring. Typical examples of test smells are excessively long code and duplicate tests [2]. While these smells are also in production code, there are test-specific smells that are based on practices inherent in test design. For example, the use of control statements in the test code can result in increased test complexity and bugs in the test code itself [2].

These general test smells focus on static aspects of test code, and their negative effects are limited to static problems such as maintainability. In this paper, we call these smells *static smells*. For instance, a test code that is too long results in difficulty in understanding the code and increased maintenance costs. However, such a code does not affect the test execution results. Similarly, test codes containing control statements indicate potential problems that can lead to test complexity and bugs in the tests themselves, which are not problematic in terms of test execution.

On the other hand, there are smells that affect the results of test execution. We call these *dynamic smells*. For example, a rotten green test [3] is a passed test that contains unexecuted assertions. Rotten green tests give developers the false impression that the test was passed without any problems, even though verification by assertions is insufficient. Dynamic smells, like the rotten green test, can lead to more serious problems than static smells. Dynamic smells cause problems with test execution results and lead to misunderstanding of test results.

Many studies have been conducted on test smells [4]–[7], and none of these studies consider the classification of test smells (static or dynamic). In addition, many tools have been proposed to detect test smells, such as TestHound [8] and tsDetect [9], and these tools detect only static smells. To our knowledge, there is no tool for dynamic smell detection.

In this paper, we introduce JTDog, a Gradle plugin for detecting dynamic smells. Although static smells can be detected only by static analysis of the test code, dynamic smells require dynamic analysis and various processes, such as test execution and coverage analysis of the test code. The user of a dynamic smell detection tool must input a great deal of information, including source code location and classpath, and a dynamic smell detection tool is difficult to use immediately. Therefore, the portability of dynamic smell detection tools tends to be lower than that of static smell detection tools. As a countermeasure to the problem of reduced portability, we can achieve high portability by incorporating a dynamic smell detection technique into the Gradle build tool. JTDog detects the three major dynamic smells: the rotten green test [3], the flaky test [10], and the dependent test [11].

In order to confirm the portability of JTDog, we applied the proposed plugin to 150 projects on GitHub. As a result, JTDog worked correctly in 122 projects and detected 958 dynamic smells in 55 of these projects.

II. JTDog

JTDog is a Gradle plugin for dynamic test smell detection for Java projects. JTDog detects three types of dynamic test smells, i.e., the rotten green test [3], the flaky test [10], and the dependent test [11], and outputs the results to a JSON file. JTDog was developed as open-source software on GitHub[1], and the JTDog plugin is available on the Gradle Plugin Portal[2].

[1]https://github.com/kusumotolab/JTDog

[2]https://plugins.gradle.org/plugin/com.github.m-tanigt.jtdog

A. Usage

JTDog provides a task labeled sniff that detects dynamic smells. Developers can detect dynamic smells present in a project by simply running the command gradle sniff. Furthermore, we can use this task by adding a single line of usage declaration to the plugins{} block in the build.gradle file, as shown below.

```
plugins {
  id 'com.github.m-tanigt.jtdog' version 'latestVersion'
}
```

B. Design

Dynamic Smell Detection Methods: In this subsection, we show the detection methods for each of three dynamic smells: the rotten green test, the flaky test, and the dependent test.

Rotten green test We follow the method of Delplanque et al. [3] to detect rotten green tests. First, we statically analyze the test code to determine the elements of each method, including assertions and method calls. Then, we run each test, and if each test is passed, then we analyze the coverage data to determine whether the test contains any unexecuted assertions.

Flaky test A flaky test is a test that both passes and fails periodically with same code. Many methods for detecting flaky tests have been proposed in recent years. One of the simplest methods, called RERUN, is to rerun a failed test multiple times and to check whether the test passes even once [10]. Other methods include monitoring changes in coverage due to code changes and classifying failed tests using machine learning [12], [13]. JTDog detects flaky tests by RERUN.

Dependent test A dependent test is a test that depends on other tests. If the test order changes, then the dependent test results will also change. A dependent test can be detected by running tests in a default order once, and then rerunning the tests in reverse order or randomly [11]. If the test results are different from those in the default order, then we can judge the test to be a dependent test. In the present study, we detect dependent tests by rerunning the test several times in random order.

Workflow: Using the detection method described in the previous section, we detect dynamic smells by the following procedure:

1) Statically analyze the test code and collect method information.
2) Embed coverage measurement instructions in the byte-code of test classes.
3) Run each instrumented test in default order.
4) If a test passes, then check whether the test is a rotten green test. If a test fails, then check whether the test is a flaky test.
5) After the tests are executed in default order, run the tests in random order to detect dependent tests.

C. Implementation

Smell Detection: In order to detect dynamic smells, JTDog first generates ASTs of test codes using Eclipse JDT and analyzes these ASTs to collect method information. JTDog then instruments test classes with JaCoCo to measure the coverages and run the tests in default order using JUnit.

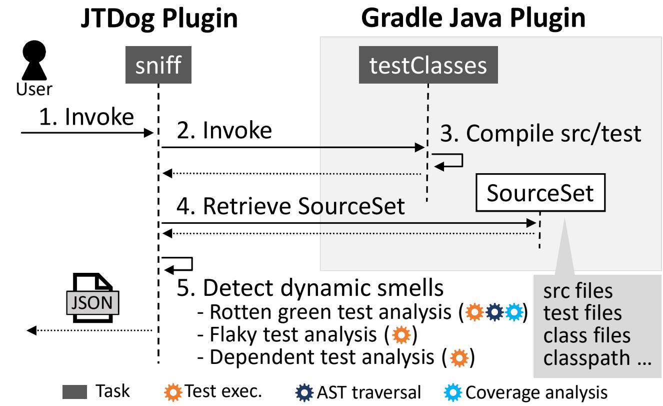

Fig. 1. Detection process using Gradle API

Second, for each test method, if the test passes, then JTDog checks whether it is a rotten green test. JTDog obtains the line numbers of the assertions that the test method contains from the method information collected in advance. By analyzing the coverage data to check whether the instruction in that line has been executed, JTDog can determine whether the test contains any unexecuted assertions. If the test includes method calls, JTDog also checks the method recursively to ascertain whether the test contains unexecuted assertions. In case of a test failure, JTDog reruns the test multiple times, and if the test passes at least once, then the test is considered to be a flaky test.

After the tests are executed in default order, JTDog performs a test execution in random order several times. Each test execution is performed on a different JVM. For each test method, the test result is compared with the result in a default order, and if the results differ, then the test is determined to be a dependent test. If the test has already been judged to be a flaky test, then the test is treated as not being a dependent test, because the test result will change independently of the order.

Using Gradle API: In addition to production code and test code, various other information, including class files and external library paths, is required in order to perform the dynamic smell detection process described above. Gradle, a build tool, can automatically manage source folders and resolve dependencies. By incorporating the dynamic smell detection technique into Gradle, all of the information necessary for execution can be retrieved from Gradle and high portability can be achieved.

Figure 1 shows an overview of the detection process using the Gradle API. In a Java project using Gradle, the Java plugin is applied to compile and test. The sniff task provided by JTDog depends on the testClasses task provided by the Java plugin. This means that when the sniff task is executed, the testClasses task is executed first and performs preprocessing, such as file compilation.

The Java plugin also introduces SourceSets into the project. A SourceSet is a logical group of Java source and resource files and has information about the compilation of source and resource files. By using a SourceSet, JTDog can retrieve source files, class files, classpath, and so on. JTDog obtains production codes and test codes from a SourceSet and collects method information by generating and analyzing the ASTs of test codes. The proposed plugin then acquires the test class files and classpath from the SourceSet, and instruments and executes tests to obtain the test results and coverage data. By combining the collected data, JTDog can detect dynamic smells.

III. Evaluation

In order to evaluate the portability of JTDog and its ability to detect dynamic smells, we applied JTDog to Gradle projects on GitHub.

A. Subjects

We selected projects on GitHub that met the following criteria:

- A project that uses the Gradle build tool, or a Maven project that can be converted to a Gradle project without problems using the conversion features of Gradle.
- The primary language is Java, and the Gradle Java plugin has been applied.
- Contains at least one test.
- Uses JUnit as the testing framework.
- Has more than 100 star ratings.

Based on the above conditions, 150 projects were selected as the target projects.

B. Experimental approach

For each project to be applied, we add a declaration for using JTDog to its `build.gradle` file, as shown in Section 2.1. Then, we run the `sniff` task and check the contents of the JSON file in which the detection results are output.

C. Results

Portability: As a result of applying JTDog to 150 selected projects, JTDog worked correctly in 122 projects, which is approximately 81% of the total number of projects. This result shows that JTDog can be applied to a variety of projects by only adding one sentence to the build file to use this plugin.

For the 28 projects that failed to apply the JTDog plugin, we visually checked for the cause of failure. The breakdown is shown in Table I. We investigated the reason for the failure due to an out of memory error, and found that this error was caused by the huge number of methods in the project. Since JTDog collects and stores method information to detect dynamic smells, the more methods there are, the more memory is used to retain the information. In other words, the insufficient heap space is the cause of the out of memory error. By increasing the heap space, we can prevent this problem. As for causes other than out of memory errors, we do not fully understand the reasons at present and need to analyze the causes of these errors in detail in the future. However, we consider that there are few problems caused by the implementation of the tool, and that JTDog has high portability.

Detection Results: Table II shows the number of dynamic smell detections in 122 projects in which JTDog worked properly and the number of projects that contains smells. Based on the results, 55 out of 122 projects have dynamic smells, which means that approximately 45% of the successfully applied projects contain at least one dynamic smell. Moreover, the rotten green test has the highest number of detections and many projects include this smell. This is because the rotten green test is a new test smell introduced in 2019 [3] and most developers are not familiar with this smell. According to Delplanque et al. [3], the rotten green test is worse than no test at all because the rotten green test gives developers the false impression that the test passed without any problems, even though the test was poorly executed. Hence, tools that can detect the rotten green test are important.

We examine the tendency of each smell in each project. Tables III, IV, and V show the number of each dynamic smell detected in each project, for the top three projects. For the rotten green test, the projects with the largest number of detections shown in Table III contained many smells with the same cause. For instance, in the traccar project with the highest number of detections, most of the 224 rotten green tests were caused by not executing assertions in `if` statements. By using JTDog, developers can learn which type of code is a rotten green test, which can be useful for future development.

Regarding the number of flaky test detections in Table IV, the number of detections in the http-request project was outstanding, but was small in the other projects. This may be

TABLE I
Causes of application failures

Cause	# of projects
Out of memory	10
Failed to create task	9
Incorrect detection result	6
Failed to read file	3

TABLE II
Detection results

Dynamic smell	# of detections	# of projects
Rotten green test	781	48
Flaky test	66	5
Dependent test	111	13
Total	958	55

TABLE III
Rotten green test detection projects

Project	#
traccar	224
elki	110
Apktool	58

TABLE IV
Flaky test detection projects

Project	#
http-request	59
java-jwt	4
spring-statem.	1

TABLE V
Dependent test detection projects

Project	#
ehcache3	57
http-request	22
micrometer	11

```java
432     @Test
433     public void c1c3IsDisposed() {
434       toV3Completable(Completable.complete())
435       .subscribe(new CompletableObserver() {
436         @Override
437         public void onSubscribe(Disposable d) {
438             //this assertion is not executed.
439             assertFalse(d.isDisposed());
440         }
441         ...
442       });
443     }
```

Fig. 2. Example of a detected rotten green test[3]

because, in addition to the fact that JTDog uses the simplest detection method, RERUN, this method is supported by many frameworks, such as Jenkins [14], and many projects deal with this smell.

In all of the top three projects in Table V, the main cause of the dependent test was sharing of static variables between tests. Such tests may change the test results by changing the test code, including adding tests that use the same static variables. It is desirable to detect dependent tests at an early stage and take countermeasures, so detection tools such as JTDog are important.

Illustrative Case: The test method c1c3IsDisposed() of the RxJavaInterop project shown in Figure 2 is a rotten green test that JTDog detected. The onSubscribe() method (line 437), an action of the event listener, has an internal assertion and is never executed. Therefore, the program is not verified by the assertion in line 439, and the test is insufficient. This kind of callback function makes difficult to confirm whether assertions will be executed or not. JTDog helps developers to notice such problems that are easily overlooked.

IV. Related Work

Many detection tools have been proposed for test smells. Breugelmans and Rompaey developed TestQ [15], a tool to quantify test smells. Greiler et al. introduced new test smells associated with test fixtures and created a detection tool, TestHound [8]. In a subsequent study, the authors extended TestHound to analyze Git and SVN repositories and released TestEvoHound [16]. Palomba et al. built TASTE [17], a tool to detect three types of test smells by textual analysis. Bavota et al. [18] created an unnamed test smells detection tool that detects nine types of test smells, and Peruma et al. proposed tsDetect [9], which can detect 19 types of test smells. All of these tools are capable of detecting only static smells and are complementary to JTDog.

Detection tools have been created for each of the three test smells that we have classified as dynamic smells. Delplanque et al. proposed a rotten green test as a new test smell and developed DrTest as a detection tool [3]. Since DrTest targets programs written in the Pharo language, Martinez et al. developed RTj [19], a rotten green test detection tool for Java programs. To detect dependent tests, Zhang et al. built DTDetector [11], and Gambi et al. built PRADET [20]. Bell et al. proposed a new detection method for flaky test and implemented DeFlaker [12]. These are all standalone tools. We integrated the detection functions of each dynamic smell into a Gradle plugin. Integration into a build tool provides high portability and allows one tool to detect all dynamic smells, which leads to increased productivity of developers.

There are integrations of test smell detection methods into IDEs. Bleser et al. developed SoCRATES [21], an IntelliJ plugin that can detect six types of test smells. Lambiase et al. created an IntelliJ plugin, DARTS [22], which can detect three types of test smells and perform automatic refactoring. Eclipse plugins have also been created. Baker et al. built TReX [23], and Santana et al. built RAIDE [24]. These plugins provide support for refactoring. Koochakzadeh and Garousi developed TeReDetect [25] and TeCReVis [26]. These tools are integrated into an Eclipse plugin called CodeCover [27]. By integrating test smell detection techniques into the IDE, they reduce the cost of test smell detection for users. Similarly, JTDog helps users to detect test smells easily by integrating dynamic test smell detection methods into Gradle.

V. Conclusions

In this paper, we introduced JTDog, a Gradle plugin for detecting dynamic test smells. In order to evaluate the portability of JTDog and its ability to detect dynamic smells, we applied JTDog to 150 projects on GitHub. JTDog detected 958 dynamic smells from 55 projects. In the future, we intend to improve the dynamic smell detection method and to extend JTDog to detect both static and dynamic smells. Moreover, we plan to enable JTDog to analyze tests from other Java testing frameworks, such as TestNG.

Acknowledgments

This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP21H04877)

References

[1] A. Deursen, L. Moonen, A. Bergh, and G. Kok, "Refactoring Test Code," in *International Conference on Extreme Programming and Flexible Processes in Software Engineering*, 2001, pp. 92–95.

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

[3] J. Delplanque, S. Ducasse, G. Polito, A. P. Black, and A. Etien, "Rotten Green Tests," in *International Conference on Software Engineering*, 2019, pp. 500–511.

[4] B. V. Rompaey, B. D. Bois, and S. Demeyer, "Characterizing the Relative Significance of a Test Smell," in *International Conference on Software Maintenance*, 2006, pp. 391–400.

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

[6] G. Bavota, A. Qusef, R. Oliveto, A. Lucia, and D. Binkley, "Are Test Smells Really Harmful? An Empirical Study," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1052–1094, 2015.

[7] A. Peruma, "What the Smell? An Empirical Investigation on the Distribution and Severity of Test Smells in Open Source Android Applications," Master's thesis, Rochester Institute of Technology, 2018.

[3] https://github.com/akarnokd/RxJavaInterop/blob/3.x/src/test/java/hu/akarnokd/rxjava3/interop/RxJavaInteropTest.java#L432

[8] M. Greiler, A. van Deursen, and M. Storey, "Automated Detection of Test Fixture Strategies and Smells," in *International Conference on Software Testing, Verification and Validation*, 2013, pp. 322–331.
[9] A. Peruma, K. Almalki, C. Newman, M. Mkaouer, A. Ouni, and F. Palomba, *tsDetect: An Open Source Test Smells Detection Tool*. Association for Computing Machinery, 2020, pp. 1650–1654.
[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An Empirical Analysis of Flaky Tests," in *International Symposium on Foundations of Software Engineering*, 2014, pp. 643–653.
[11] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. Ernst, and D. Notkin, "Empirically Revisiting the Test Independence Assumption," in *International Symposium on Software Testing and Analysis*, 2014, pp. 385–396.
[12] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "DeFlaker: Automatically Detecting Flaky Tests," in *International Conference on Software Engineering*, 2018, pp. 433–444.
[13] K. Herzig and N. Nagappan, "Empirically Detecting False Test Alarms Using Association Rules," in *International Conference on Software Engineering*, 2015, pp. 39–48.
[14] Jenkins RandomFail annotation, "@RandomlyFails," https://github.com/jenkinsci/jenkins-test-harness/blob/master/src/main/java/org/jvnet/hudson/test/RandomlyFails.java, 2017.
[15] M. Breugelmans and B. V. Rompaey, *TestQ: Exploring Structural and Maintenance Characteristics of Unit Test Suites*, International Workshop on Advanced Software Development Tools and Techniques, 2008.
[16] M. Greiler, A. Zaidman, A. van Deursen, and M. Storey, "Strategies for avoiding text fixture smells during software evolution," in *Working Conference on Mining Software Repositories*, 2013, pp. 387–396.
[17] F. Palomba, A. Zaidman, and A. D. Lucia, "Automatic Test Smell Detection Using Information Retrieval Techniques," in *International Conference on Software Maintenance and Evolution*, 2018, pp. 311–322.
[18] G. Bavota, A. Qusef, R. Oliveto, A. D. Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *International Conference on Software Maintenance*, 2012, pp. 56–65.
[19] M. Martinez, A. Etien, S. Ducasse, and C. Fuhrman, "RTj: A Java Framework for Detecting and Refactoring Rotten Green Test Cases," in *International Conference on Software Engineering*, 2020, pp. 69–72.
[20] A. Gambi, J. Bell, and A. Zeller, "Practical Test Dependency Detection," in *International Conference on Software Testing, Verification and Validation*, 2018, pp. 1–11.
[21] J. D. Bleser, D. D. Nucci, and C. D. Roover, "SoCRATES: Scala Radar for Test Smells," in *Symposium on Scala*, 2019, pp. 22–26.
[22] S. Lambiase, A. Cupito, F. Pecorelli, A. D. Lucia, and F. Palomba, "Just-In-Time Test Smell Detection and Refactoring: The DARTS Project," in *International Conference on Program Comprehension*, 2020, pp. 441–445.
[23] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiss, "TRex - The Refactoring and Metrics Tool for TTCN-3 Test Specifications," in *Testing: Academic & Industrial Conference - Practice And Research Techniques*, 2006, pp. 90–94.
[24] R. Santana, L. Martins, L. Rocha, T. Virgínio, A. Cruz, H. Costa, and I. Machado, "RAIDE: A Tool for Assertion Roulette and Duplicate Assert Identification and Refactoring," in *Proceedings of the 34th Brazilian Symposium on Software Engineering*, 2020, pp. 374–379.
[25] N. Koochakzadeh and V. Garousi, "A Tester-Assisted Methodology for Test Redundancy Detection," *Advances in Software Engineering*, vol. 2010, 2010.
[26] N. Koochakzadeh and V. Garousi, "TeCReVis: A Tool for Test Coverage and Test Redundancy Visualization," in *Testing – Practice and Research Techniques*, 2010, pp. 129–136.
[27] CodeCover, http://codecover.org/index.html.