tsDetect: An Open Source Test Smells Detection Tool

Anthony Peruma
axp6201@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Khalid Almalki
ksa8566@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Christian D. Newman
cnewman@se.rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Mohamed Wiem Mkaouer
mwmvse@rit.edu
Rochester Institute of Technology
Rochester, New York, USA

Ali Ouni
ali.ouni@etsmtl.ca
ETS Montreal, University of Quebec
Montreal, Quebec, Canada

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

ABSTRACT

The test code, just like production source code, is subject to bad design and programming practices, also known as smells. The presence of test smells in a software project may affect the quality, maintainability, and extendability of test suites making them less effective in finding potential faults and quality issues in the project's production code. In this paper, we introduce tsDetect, an automated test smell detection tool for Java software systems that uses a set of detection rules to locate existing test smells in test code. We evaluate the effectiveness of tsDetect on a benchmark of 65 unit test files containing instances of 19 test smell types. Results show that tsDetect achieves a high detection accuracy with an average precision score of 96% and an average recall score of 97%. tsDetect is publicly available, with a demo video, at: https://testsmells.github.io/

KEYWORDS

Software Quality, Test Smells, Detection Tool

ACM Reference Format:
Anthony Peruma, Khalid Almalki, Christian D. Newman, Mohamed Wiem Mkaouer, Ali Ouni, and Fabio Palomba. 2020. tsDetect: An Open Source Test Smells Detection Tool. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Quality assurance is a crucial driver in any software project, and the success of the project depends on the quality aspects it exhibits, meaning we need to optimize these aspects. Software testing is one of the widely-acknowledged techniques that are of paramount importance for quality assurance. There are different types of testing strategies that can be adopted to test a software system, such as unit testing [27]. Unit testing involves developers testing the smallest unit in the production codebase, which is typically a method. Hence, this process requires developers writing test code to exercise production code.

Even though the unit test code will not be executed in production, it is essential that developers follow standard programming practices when implementing a test suite. This practice ensures that the production code is effectively exercised and enables developers to easily identify defects in the system. Similar to traditional code smells, *i.e.*, bad programming practices, unit tests may also suffer from smells that are exclusive to the testing domain [16]. Furthermore, just like how traditional code smells cause a system to be more prone to changes and defects [20], it has been shown that test smells may negatively impact the quality of the system [30].

Therefore, there is a growing need to incorporate the verification of bad testing practices into modern code reviews. While there exist open-source quality assurance tools, such as PMD [3], Checkstyle [1], and FindBugs [2], which primarily focus on detecting a variety of quality issues in production code, the number of open-source tools that detect a wide variety of test smells and supports integration with continuous integration frameworks is limited.

This paper introduces tsDetect, an open-source tool that currently supports the detection of 19 common test smells, *i.e.*, deviations from good unit testing programming practices, as advocated in xUnit guidelines [16, 17]. These 19 smells are part of the catalog of unit test smells, and details about their definitions, rationale, and examples appear in published literature [22]. tsDetect takes, as input, software project source code and first separates the set of unit test files from production source files, then generates their Abstract Syntax Trees (ASTs) in order to search for any predefined patterns of bad test programming practices syntactically using detection rules. tsDetect generates, as output, a file containing all detected violations. tsDetect has been designed to be easy to extend, *i.e.*, developers can easily calibrate the predefined rules and add their own customized rules if needed. Moreover, although tsDetect currently detects 19 test smells, it is designed with a high level of flexibility to incorporate new smell types easily, and also permits the customization of the existing smell detection rules as shown later in Section 3, where we discuss the architecture of the tool.

To evaluate the correctness of tsDetect, we performed a qualitative analysis on a benchmark of 65 unit test files that contain instances from various smell types. Analysis of our tool has shown that tsDetect is able to correctly detect test smells with a precision score ranging from 85% to 100% and a recall score from 90% to 100% with an average F-score of 96.5%.

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Furthermore, we have utilized tsDetect to perform a large-scale empirical study on open-source Android applications (apps) and traditional Java systems [22]. Our study investigated the occurrences and distribution of the 19 test smells on 656 open-source Android apps, including a comparison of smell type occurrence with traditional Java systems. In a subsequent study [26], we explored the impact and relationship between refactoring operations and test smells, detected by tsDetect, in 250 open-source Android apps.

Open source tool and documentation. The tsDetect project's website [4] includes the tool source code, along with a demonstration video and documentation on how to use it. The website also contains examples and general information about test smells. Additionally, tsDetect has received increasing interest on GitHub from the development and research communities in the form of forks and feedback on how to improve our tool and make it more practical. We encourage the community to contribute improvements and extensions to tsDetect.

2 BACKGROUND AND MOTIVATION

Test smells represent deviations from well-established testing practices and guidelines on how test cases should be designed, implemented, and how they should interact with each other. The detection of such potential deviations is typically a tedious, manual, and error-prone task for developers and testers. Effective detection of test smells requires expertise from developers to inspect, understand, and run the test to be able to correctly detect the violations.

The goal of implementing tsDetect is to provide developers with an automated approach for optimizing the quality of their test suites. tsDetect has the ability to detect 19 smells occurring in JUnit based unit test files, some of which have been highlighted in existing literature as being problematic [9, 17, 30, 32]. The list of detected smells supported by tsDetect is reported in Table 1 along with their definitions/detection rules. In summary, tsDetect analyzes the test suite for the existence of certain violations to xUnit testing guidelines [16, 17]. Even though all test smells detected by the tool apply to any Java-based system, one smell, namely Default Test, is specific to Android applications. For the sake of space limitations, we provide the necessary background information about the smells supported by tsDetect in our prior study [22]. Additionally, our project's website [4] contains examples of real-world code snippets that exhibit each of the supported smell types.

The detected test smells by tsDetect provides valuable support for developers. Our prior work [22] has shown the prevalence of these smells in open-source systems, indicating the difficulty of developers to remove them from the codebase during the lifetime of the project. Furthermore, in our previous qualitative analysis [22] in which we reached out to developers whose test files exhibit a test smell detected by tsDetect, most of the surveyed developers confirmed that the test files identified by tsDetect did contain instances of bad unit testing programming practices. Additionally, based on the output of our tool, some of these developers made necessary corrections to their code based on the findings. Hence, to ensure that unit tests are of high-quality and maintenance-friendly, we provide the community with a tool that can be integrated into the development workflow to automate the identification of bad unit testing practices early during the system's development lifecycle.

3 TSDETECT ARCHITECTURE

tsDetect is implemented as an open-source, command line-based tool that is available as a standalone Java `jar` file. By providing tsDetect as a self-contained executable file rather than a plugin (which is part of our future work), users are not required to have a specific Integrated Development Environment (IDE) installed on their machine in order to detect smells in their test code. Similar to other code smell and defect detection tools such as PMD and FindBugs, offering tsDetect as an executable through the command line facilitates its integration with modern continuous integration frameworks, as well as its adoption in mining software repositories and empirical studies in software engineering.

In addition to the tsDetect detection mechanism, we incorporate supplementary modules to automate the entire detection workflow. These modules support the detection process by parsing the input source files to detect unit test files (and their corresponding production files) in the project hierarchy. A high-level overview of the architecture of tsDetect is depicted in Figure 1. In ① and ②, the test and production files are identified from the project structure. In ③ and ④, tsDetect checks if the test files exhibits test smells. In ⑤, the results from the test smell detection process are saved. In the following subsections, we describe the detector and the mechanism to distinguish test files and their corresponding production files.

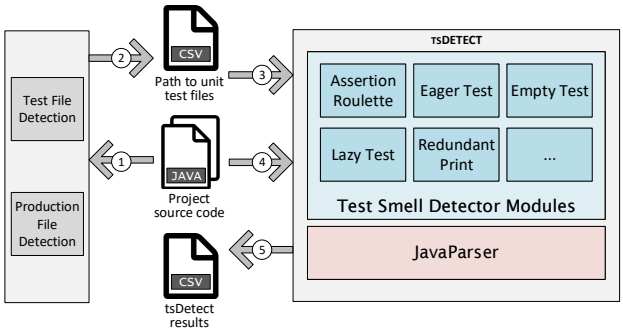

Figure 1: High-level architecture of tsDetect

3.1 Test Smell Detection

We followed a strategy design pattern in implementing the detection mechanism for test smells (UML class diagrams are available on the project website). Each smell is implemented and runs independently of other smells. The detection strategy of each smell type is self-contained within its own module. This design pattern also enables the seamless addition of new smell detectors in the future. Internally, tsDetect calls the JavaParser [12] library to parse the source code files. JavaParser builds an AST from the unit test file that is under analysis. The AST is then analyzed by each of the available smell detection modules based on the detection rules defined in Table 1. Depending on the type of smell being detected, we override the appropriate `visit()` method to perform the detection. For example, in the case of detecting the *Redundant Print* smell, we first create a `MethodDeclaration` visitor to identify all test methods in the class. Next, for each detected test method, we create a `MethodCallExpr` visitor to examine the methods being called within the test method. Finally, for each called method, we check if the name of the method matches a Java print method to determine if the file is smelly.

Table 1: Summary of the test smell detection rules built-in tsDetect

Test Smell	Detection Rule
Assertion Roulette	A test method contains more than one assertion statement without an explanation/message (parameter in the assertion method)
Conditional Test Logic	A test method that contains one or more control statements (i.e., if, switch, conditional expression, for, foreach and while statement)
Constructor Initialization	A test class that contains a constructor declaration
Default Test	A test class is named either 'ExampleUnitTest' or 'ExampleInstrumentedTest'
Duplicate Assert	A test method that contains more than one assertion statement with the same parameters
Eager Test	A test method contains multiple calls to multiple production methods
Empty Test	A test method that does not contain a single executable statement
Exception Handling	A test method that contains either a throw statement or a catch clause
General Fixture	Not all fields instantiated within the `setUp` method of a test class are utilized by all test methods in the same test class
Ignored Test	A test method or class that contains the `@Ignore` annotation
Lazy Test	Multiple test methods calling the same production method
Magic Number Test	An assertion method that contains a numeric literal as an argument
Mystery Guest	A test method containing object instances of files and databases classes
Redundant Print	A test method that invokes either the `print` or `println` or `printf` or `write` method of the `System` class
Redundant Assertion	A test method that contains an assertion statement in which the expected and actual parameters are the same
Resource Optimism	A test method utilizes an instance of a `File` class without calling the method `exists()`, `isFile()` or `notExists()` methods of the object
Sensitive Equality	A test method invokes the `toString()` method of an object
Sleepy Test	A test method that invokes the `Thread.sleep()` method
Unknown Test	A test method that does not contain a single assertion statement and `@Test(expected)` annotation parameter

On completion, the results are saved into a Comma-Separated Values (CSV) file. For each smell type, tsDetect outputs a boolean value indicating if the smell is present or not in the file. We decided on a CSV format for output as this format is technology independent and permits users to import the data into a database system of their choice for ease of analysis.

Test File Detection. JUnit recommends that developers follow the naming convention [13] of either pre-pending or appending the word 'Test' to the name of the production file that is to be tested (*i.e.*, Test*.java and *Test.java). Our tool first identifies all '.java' files where the filename either starts or ends with the word 'test'. Next, for each of the identified Java source files, the tool parses its AST using JavaParser. The purpose of using the AST is twofold. First, we are able to eliminate Java files that contain syntax errors, and secondly, we are able to accurately detect if the file contained JUnit-based unit test methods. For a file to contain a unit test method, the method should have a `public` access modifier, and either has an annotation called `@Test` (JUnit 4) or the method name should start with 'test' (JUnit 3).

Production File Detection. In order to detect some test smells, *e.g., Eager Test* and *Lazy Test*, the production file associated with the unit test file is required. To identify the production file, we explore the project structure to search for files that have the same name as the test file, but without the word 'test'. Next, for each production file we identified, the tool generates its corresponding AST to ensure that the file is syntactically correct.

tsDetect Usage. As a command line tool, *tsDetect* can be executed via the following command:

```
java -jar .\TestSmellDetector.jar <path_to_test_files>
```

Once tsDetect has been started, it requires no further user intervention. After the detection process is completed, a CSV file containing the results of the detection process will be created and returned as output.

4 TSDETECT APPLICABILITY

Practitioners. We envision developers integrating tsDetect into their development toolset and workflow. For example, by integrating it into their build process, developers will be notified of smells in their test code before either committing the file into the repository or generating a production release. Furthermore, in our previous work [22], we reported smells detected by tsDetect to 120 developers, from various open-source systems. We received responses from 50 developers, with most developers confirming that the programming constructs we highlighted in their code are indeed examples of test smells, and that they will take corrective actions to fix them.

Researchers. With the capability of batch-based analysis, tsDetect can be used by researchers in empirical studies on software quality and maintenance. As previously stated, we have utilized tsDetect in empirical studies of open-source systems [22, 26]. Additionally, tsDetect has also been utilized by the research community in the study of test smells. Schvarcbacher et al. [29] integrate tsDetect into a code quality monitoring system to study reactions to test smells. Spadini et al. [31] use tsDetect in their study of severity rating of test smells and their impact on the maintainability of test suites. In a study on the evolution of test smells, Kim [14] uses tsDetect to show that test smells persist in systems even though developers refactor the code.

Educators. tsDetect can also be used by software engineering educators in the classroom to teach students the importance of designing high quality, maintenance-friendly test suites. As an example, tsDetect is being used as part of an activity in a software testing course offered by the Department of Software Engineering at Rochester Institute of Technology[1].

5 EVALUATION

We conducted an empirical study on the effectiveness of tsDetect in correctly detecting test smells, in terms of precision and recall. As there are no existing datasets containing information for all the supported smells, we decided to construct our own validation set.

[1]https://www.se.rit.edu/

We start by randomly selecting test files (and their corresponding production files) from 656 open-source Android apps, hosted on F-Droid [8]. We ensured that these apps were not duplicated or forked by verifying the uniqueness of the source URL and commit SHA. Due to space constraints, we provide the details of these projects on our website [4]. We then use the definition of test smells to identify them in the source files. Upon the identification of smells in a test file, we tag it along with its corresponding production file and the types of smells its exhibits. We keep this process of manually analyzing files and annotating them until we reach 20 infected instances per smell. This process resulted in a total of 65 annotated files.

Next, to ensure an unbiased annotation process, we performed a manual analysis by involving another set of reviewers to review the existing annotated set. We involved 39 graduate and undergraduate students from the Department of Software Engineering at Rochester Institute of Technology to manually review the same files for the existence of test smells. All participants volunteered to participate in the experiment and were familiar with Java programming, unit testing, and quality assurance concepts. The experience of these participants with Java development ranged from 2 to 11 years, which includes exposure to developing unit tests. Prior to the review process, the participants were provided with a 75-minute tutorial on test smells along with reference materials. To reduce the effect of bias, participants were randomly grouped into groups of three, resulting in a total of 13 groups. Each group was provided with ten test files. The number of smell types exhibited by each file ranged from one to six. On average, each file contained three smell types. However, the participants were not informed of the type or count of smells contained in their set of test files.

To further protect from bias, we provided each smell type to at least two groups. Each group was asked to annotate each of the assigned files with the smell they think it contains. The participants were offered a period of three days to submit their survey results. When reviewing the survey results, we noticed two cases from the same group where there was no agreement about the existence of a *Mystery Guest*. There was no consensus between the group members on whether they should consider a service API call a guest in part because it is not mentioned in the provided definition. Therefore, we decided to filter out these two cases, and we replaced them with two other manually identified mystery guests which were reviewed again by the group. This process generated a revised annotated set that we use as our oracle for testing the detection accuracy. We next ran our tool on the same set of test files and then compared our results against the oracle. For each smell type, we constructed a confusion matrix and calculated the precision, recall, accuracy, and F-Score.

Table 2 reports the detection results for each smell type. As shown in the table, tsDetect achieves a high level of correctness with F-Scores ranging from 87.8% to 100%. For the cases where the tool did not achieve 100%, we investigate the slight mismatch between the tool and the human decision. We had only one case where our tool did not detect a smell, and we had to refine our detection rule. The other cases were related to the human interpretation of the smell definition, and that is why developers can update the default rules to better match what they consider as problematic. The dataset used in this experiment is available on our website [4].

Table 2: Test smell detection correctness of tsDetect

Smell Type	# smell instances	# detected | corrected instances	Precision	Recall	F-Score
Assertion Roulette	20	19 | 18	94.74%	90.00%	92.31%
Conditional Test Logic	20	20 | 20	100%	100%	100%
Constructor Initialization	20	20 | 20	100%	100%	100%
Default Test	20	20 | 20	100%	100%	100%
Duplicate Assert	20	21 | 18	85.71%	90.00%	87.80%
Eager Test	20	20 | 20	100%	100%	100%
Empty Test	20	20 | 20	100%	100%	100%
Exception Handling	20	20 | 20	100%	100%	100%
General Fixture	20	21 | 20	95.24%	100%	97.56%
Ignored Test	20	20 | 20	100%	100%	100%
Lazy Test	20	22 | 20	90.91%	100%	95.24%
Magic Number Test	20	20 | 20	100%	100%	100%
Mystery Guest	20	20 | 19	95.00%	95.00%	95.00%
Redundant Print	20	20 | 20	100%	100%	100%
Redundant Assertion	20	23 | 20	85.96%	100%	93.02%
Resource Optimism	20	20 | 20	100%	100%	100%
Sensitive Equality	20	20 | 18	90.00%	90.00%	90.00%
Sleepy Test	20	18 | 18	100%	90.00%	94.74%
Unknown Test	20	21 | 18	85.71%	90.00%	87.80%
Average	–	–	**96.01%**	**97.11%**	**96.50%**

6 RELATED WORK

Research on test smells has proposed test smells, conducted empirical studies on projects related to test smells, and proposed tools for the detection of test smells. In this section, we focus only on peer-reviewed studies that are related to test smell detection tools.

Breugelmans et al. [7] built a tool, TestQ, which allows developers to visually explore test suites and quantify test smells. Similarly, Koochakzadeh et al. [15] built a Java plugin, TeCRevis, for the visualization of redundant tests. Neukirchen et al. [18] created T-Rex, a tool that detects violations of test cases to the Testing and Test Control Notation (TTCN-3). In other studies, Greiler et al. [9] introduced new test smells related to test fixtures and also built a detection tool, TestHound, as part of their research. In a subsequent study [10], the authors extended the tool to mine Git and SVN repositories for test fixture smells. Reichhart et al. [28] proposed a tool for the detection of test smells in Smalltalk. Zang et al. [33] built DTDetector to detect dependent tests. Bavota et al. [6] built a tool that can detect nine types of test smells, while Palomba et al. [21] devised a tool for detecting three types of test smells by means of textual analysis.

7 CONCLUSION AND FUTURE WORK

In this paper, we introduced tsDetect, an open-source tool that can detect 19 types of test smells occurring in JUnit-based test suites. We also described the architecture of tsDetect, the ease of integrating new smell types into the tool, and its use in research studies by the community. To evaluate our tool, we conducted a set of experiments on the soundness of tsDetect. The results show that our tool achieves a high performance in terms of F-score with an average of 96.5% on all the considered test smells types.

Our future work includes improvements/extensions to the tool. The next step will be to integrate tsDetect with common IDEs in the form of a plugin. We aim to extend tsDetect to new types of test smells, such as those showing up in the naming practices of the test. This will leverage existing research on naming practices [5, 11, 19, 23–25] to study tests and determine when these practices are smelly. Finally, we encourage the community to download tsDetect and contribute to its improvement and extension.

REFERENCES

[1] Checkstyle. https://checkstyle.org/.
[2] Findbugs. http://findbugs.sourceforge.net/.
[3] Pmd. https://pmd.github.io/.
[4] tsdetect. https://testsmells.github.io/.
[5] V. Arnaoudova, M. Di Penta, and G. Antoniol. Linguistic antipatterns: What they are and how developers perceive them. *Empirical Softw. Engg.*, 21(1):104–158, Feb. 2016.
[6] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley. Are test smells really harmful? an empirical study. *Empirical Software Engineering*, 20(4):1052–1094, 2015.
[7] M. Breugelmans and B. V. Rompaey. Testq: Exploring structural and maintenance characteristics of unit test suites. In *International workshop on advanced software development tools and Techniques*, 2008.
[8] F-Droid. https://f-droid.org/.
[9] M. Greiler, A. van Deursen, and M.-A. Storey. Automated detection of test fixture strategies and smells. In *International Conference on Software Testing, Verification and Validation*, pages 322–331, 2013.
[10] M. Greiler, A. Zaidman, A. v. Deursen, and M.-A. Storey. Strategies for avoiding text fixture smells during software evolution. In *Working Conference on Mining Software Repositories*, pages 387–396, 2013.
[11] E. W. Høst and B. M. Østvold. Debugging method names. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 294–317, Berlin, Heidelberg, 2009. Springer-Verlag.
[12] JavaParser. https://javaparser.org/.
[13] JUnit. FAQ. https://junit.org/junit4/faq.html#running_15/.
[14] D. J. Kim. An empirical study on the evolution of test smell. In *Proceedings of the 42nd International Conference on Software Engineering: Companion Proceedings*, ICSE âĂŹ20, 2020.
[15] N. Koochakzadeh and V. Garousi. Tecrevis: a tool for test coverage and test redundancy visualization. *Testing–Practice and Research Techniques*, pages 129–136, 2010.
[16] G. Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.
[17] G. G. Meszaros. Xunit test patterns and smells: Improving the roi of test code. In *International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, 2010.
[18] H. Neukirchen and M. Bisanz. Utilising code smells to detect quality problems in ttcn-3 test suites. *Testing of Software and Communicating Systems*, pages 228–243, 2007.
[19] C. D. Newman, A. Peruma, and R. AlSuhaibani. Modeling the relationship between identifier name and behavior. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2019.
[20] 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*, 23(3):1188–1221, 2018.
[21] F. Palomba, A. Zaidman, and A. De Lucia. Automatic test smell detection using information retrieval techniques. In *International Conference on Software Maintenance and Evolution*, pages 311–322, 2018.
[22] A. Peruma, K. Almalki, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. On the distribution of test smells in open source android applications: An exploratory study. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering*, CASCON '19, pages 193–202, Riverton, NJ, USA, 2019. IBM Corp.
[23] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. Contextualizing rename decisions using refactorings, commit messages, and data types. *Journal of Systems and Software*.
[24] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. An empirical investigation of how and why developers rename identifiers. In *International Workshop on Refactoring 2018*, 2018.
[25] A. Peruma, M. W. Mkaouer, M. J. Decker, and C. D. Newman. Contextualizing rename decisions using refactorings and commit messages. In *Proceedings of the 19th IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2019.
[26] A. Peruma, C. D. Newman, M. W. Mkaouer, A. Ouni, and F. Palomba. An exploratory study on the refactoring of unit test files in android applications. In *Proceedings of the 4th International Workshop on Refactoring*, IWoR 2020, New York, NY, USA, 2020. Association for Computing Machinery.
[27] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, USA, 7 edition, 2010.
[28] S. Reichhart, T. Gîrba, and S. Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.
[29] M. Schvarcbacher, D. Spadini, M. Bruntink, and A. Oprescu. Investigating developer perception on test smells using better code hub-work in progress. In *2019 Seminar Series on Advanced Techniques and Tools for Software Evolution, SATTOSE 2019*, 2019.
[30] D. Spadini, F. Palomba, A. Zaidman, M. Bruntink, and A. Bacchelli. On the relation of test smells to software code quality. In *International Conference on Software Maintenance and Evolution*, pages 1–12, 2018.
[31] D. Spadini, M. Schvarcbacher, A.-M. Oprescu, M. Bruntink, and A. Bacchelli. Investigating severity thresholds for test smells. In *Proceedings of the 17th International Conference on Mining Software Repositories*, MSR âĂŹ20, 2020.
[32] 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 *Int. Conf. on Autom. Software Engineering*, pages 4–15, 2016.
[33] S. Zhang, D. Jalali, J. Wuttke, K. Muşlu, W. Lam, M. D. Ernst, and D. Notkin. Empirically revisiting the test independence assumption. In *Int. Symposium on Software Testing and Analysis*, pages 385–396, 2014.

APPENDIX

As requested by the submission guidelines, we are providing an Appendix that contains a walkthrough of using tsDetect and other related sources/artifacts.

Source Code

The source code for tsDetect is available in a publicly accessible GitHub repository: https://github.com/TestSmells/TestSmellDetector

Project Website

In addition to resources around the 19 test smells, our project website contains details around the studies that we have performed using tsDetect, including links to the relevant datasets. Additionally, we also maintain a running list of research publications that use tsDetect. The project website is available at: https://testsmells.github.io

Demonstration Video

A video demonstration on how to use tsDetect to identify test smells in JUnit based unit test files is available on YouTube (https://www.youtube.com/watch?v=kzRSadHo5YA) and also on our project website.

tsDetect Walkthrough

(1) Download the latest release from: https://github.com/TestSmells/TestSmellDetector/releases

(2) Before executing the tool, a CSV file needs to be created. The CSV file specifies the list of test files (and their associated production file). This file will be used as input to the tool. The format of the file should be:
`appName,pathToTestFile,pathToProductionFile`
Example:
`myCoolApp,F:\Apps\myCoolApp\code\test\GraphTest.java,F:\Apps\myCoolApp\code\src\Graph.java`
`myCoolApp,F:\Apps\myCoolApp\code\test\EmployeeTest.java,F:\Apps\myCoolApp\code\src\Employee.java`
`myCoolApp,F:\Apps\myCoolApp\code\test\EmployeeRelationship.java`
Note: In the event a production file is not associated with a test file, then detection for test smells that require production files are not run

(3) Once the CSV file has been created, the path to the CSV file needs to be passed as an argument when executing the jar. The format of the file should be:
`java -jar .\TestSmellDetector.jar pathToInputFile.csv`
Example:
`java -jar .\TestSmellDetector.jar "F:\Projects\TestSmellDetector\inputFile.csv"`

(4) The tool outputs a CSV file containing the results of the execution. The output CSV file will be created in the same location as the jar. The CSV file contains the path of the test files (and their associated production file) along with the detection status for each smell. A detection status of 'true' indicates that the associated smell exists in the test file.