2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion) | 978-1-6654-9598-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSE-Companion55297.2022.9793823

Behavior-based test smells refactoring

Toward an automatic approach to refactoring Eager Test and Lazy Test smells

Adriano Pizzini
Graduate Program in Computer Science – PPGIa - PUCPR
Curitiba, Paraná Brazil
adriano.pizzini@ppgia.pucpr.br

ABSTRACT

Software testing is an essential part of the development process, and like many software artifacts, tests are affected by smells, harming comprehension and maintainability. Several studies are related to test smell identification, but few studies are related to refactoring. Most proposed approaches are semi-automated, with the developer as a safety net. This paper presents a proposal for automatic refactoring of Eager Test and Lazy Test smells based on identifying the behavior of tests and, consequently, the behavior of the System Under Test (SUT). The approach will be evaluated with private source code repositories to identify its impact on quality attributes.

KEYWORDS

Test smell refactoring, Software Quality, Testing.

ACM Reference format:

Adriano Pizzini. 2022. Behavior-based test smells refactoring: Towards an automatic approach to refactoring Eager Test and Lazy Test smells. In *Proceedings of IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion). Pittsburgh, PA, USA, 2 pages.* https://doi.org/10.1145/3510454.3517059

1. Introduction

Unit testing is a common practice, in which developers write test cases together with regular code [1]. Such as other artifacts, unit tests are affected by quality problems known as Test Smells (TS). Test smells are indicators or symptoms of more severe design problems [2, 3], created from poor or suboptimal solutions [4, 5, 6] that violate good practices in a given domain [7]. Smells hinder the evolution of the software [2, 4], impact product quality [7, 8, 9], contribute to reduce developers' productivity. Consequently, they increase the time required to understand the programs and the possibility of failures [9, 10, 11, 12].

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM 978-1-6654-9598-1/22/05...$15.00
https://doi.org/10.1145/3510454.3517059

A potential solution is allocating resources like time and budget in refactoring [13], but [14] showed that the development team may not identify test smells and might unintentionally introduce TS in refactoring operations due to a lack of skills. Therefore, tools like TS identifiers and refactoring helpers may support the development team in refactoring operations [15, 16]. Many studies have proposed detection tools [15] or evaluated how TS impacts development processes with respect to aspects such as maintenance and comprehension [2, 12, 13, 14]. Few studies have presented approaches for refactoring TS [15] and, the proposed approaches are human-dependent, requiring the developer to be present in the process. However, resource limitations, time constraints and lack of knowledge hinder these human-dependent approaches. Thus, in this paper, we propose an approach for automated refactoring unit testing to remove Eager Test (ET) and Lazy Test (LT) smells. ET is one of the most significant smells related to change-proneness [11], and ET and LT are the most prevalent smells in Scala projects [17].

2. Test smell refactoring

Refactoring has become a well-established and disciplined Software Engineering (SE) practice that has attracted a significant amount of research, motivated by the need to improve system structures [18].

Test refactoring should preserve the behavior of the tests (test scenario) while no errors are introduced into the test suite [16]. Preserving the test's behavior implies preserving the behavior of the System Under Test (SUT), i.e., the initial state of the SUT (the setup), how it reacts to each test, and how internal state modifications are made. As tests may be executed in a random or predefined sequence, the refactoring operation must analyze the context of the test suite. Unlike semi-automated approaches, which rely on developer approving of the refactoring proposal, automated approaches must ensure that the TS is removed while leaving the behavior of the SUT and tests intact.

Test scenario analysis is necessary because a developer places the SUT under a specific state, which one or more tests may use. After the setup phase, a set of tests is executed by one or more methods, and one test may modify the internal behavior of the SUT, which impacts the execution of the following tests.

3. Proposed approach

The proposed approach is composed by four main steps: (1) Instrumenting test and source code, (2) Identifying the behavior, (3) Finding compatible tests, and (4) Refactoring, explained below.

(1) Instrumenting: consists of instrumenting the SUT source code and tests to identify the entry and exit points of the methods, modifications in the attributes of the SUT classes, selection, and repetition structures. The instrumentation will allow the creation of the code execution tree, from which the behavior of the tests and the SUT can be identified. Since instrumentation depends on the syntactic and semantic analysis of the test code and the SUT, in this same step, the identification of specific points of the code will be carried out, such as the creation of objects and modifications in the internal states of the created objects. The identification of these specific points is related to the use of stereotypes, such as those proposed by [21]. Stereotypes allow you to classify code structures into generic elements that reduce the size of the code execution tree. For example, the creation of a SUT object in the test class can be represented by the "Constructor" stereotype. In contrast, the execution of a SUT method that modifies the internal state of the SUT object can be represented by the "Set" or "Command" stereotypes. In the case of executing methods that do not change the internal state of SUT objects, stereotypes such as "Get", "Predicate", and "Property" [21] can be used. Furthermore, identifying these stereotypes within the SUT helps identify how objects in the SUT interact with each other, leading to a better comprehension of their behavior.

(2) Identifying the behavior: consists of identifying the behavior of the tests and the SUT by executing the tests and collecting execution trace information to create the code execution tree. This step provides a safeguard against hidden data that may change the behavior, such as databases and external files, that do not appear in static source code analysis. For example, a test method that contains two tests based on the same setup whose tests do not change the state of the SUT object can be refactored by simply extracting the methods. However, suppose there is a change in the state of the SUT object after running the first test. In that case, it is necessary to preserve the order of execution of these tests so that the execution of the refactored methods does not change the order of execution of the methods and, consequently, the SUT behavior. Also, it is necessary to identify which lines of code are associated with each test.

(3) Identifying compatible tests: the identification of compatible tests can be performed through meta-heuristics such as tabu search. We consider two tests compatible if they test the same SUT objects with the same states. As we propose to refactor ET and LT, it is necessary to investigate all test classes in the repository to find compatible tests.

(4) Refactoring: consists of applying the extraction refactoring operation in a smelly test and the consequent reordering of the tests so that the test order does not affect the SUT's behavior. The first step to refactor tests is to verify that the setup is isolated from the other lines of code. If the setup is part of the test method (i.e., not isolated), it will need to be extracted to a specific method so that it can be shared by two or more smelly tests, avoiding code duplication. Subsequently, the tests are extracted and placed into separate test methods, parameterized to run in the same pre-refactoring order. Also, it is necessary to identify, line-by-line, which parts of test code must be extracted to each test.

3.1. Related work

Lambiase et al. [19] is the closest work to our approach. Their proposal uses textual detection rules from [20], to propose a semi-automated plugin-based approach to help developers detect and refactor General Fixture, Eager Test, and Lack of Cohesion TS at commit level. Their detection mechanism was empirically evaluated and is based on similarity of method names, which can produce problems in the detection because the same method can be executed more than once by a test with changes in the parameters, which change the flow of execution of the SUT code. Furthermore, refactoring evaluation was not done. According to [19], "*DARTS implements refactoring action that are supposed to only improve quality aspects of test code without altering the way it exercises production classes*".

However, their proposal does not consider aspects related to testing scenarios, such as setup used, order of execution of the test methods, and states of the SUT objects used to verify that test coverage remains unchanged. In addition to test scenario analysis, we propose incorporating Lazy Test smell into refactoring and evaluating the proposed approach in the test repository.

4. Evaluation plan

We will conduct controlled experiments [22] to refactor a test suite and collect data from both original and refactored tests to evaluate the proposed approach. To evaluate reliability, we plan to use code coverage. The metrics of Buse and Weimer [23] and the model of Postnett et al. [24] can assess readability.

To evaluate internal attributes and maintainability [25], we plan to use a set of metrics: Afferent connections of a class (ACC), Average cyclomatic complexity per method (ACCM), Average lines of code per method (AMLOC), Coupling between Objects (CBO), Coupling Factor (CF), Change Dependency Between Classes (CDBC), McCabe Cyclomatic Complexity (CC), Message Passing Coupling (MPC), Response for a Class (RFC), and Weighted Method Count (WMC). In addition to these metrics, we intend to define some specific metrics to evaluate test code refactoring, such as Number of setups (NoS), Number of tests (NoT), and Reuse of setups (RoS).

ACKNOWLEDGMENTS

A. Pizzini thanks Pontifical Catholic University of Paraná (PUCPR) and Federal Institute Catarinense (IFC) for the financial support.

REFERENCES

[1] Daka, Ermira; Fraser, Gordon. A Survey on Unit Testing Practices and Problems. 2014 IEEE 25th International Symposium on Software Reliability Engineering.

[2] Yamashita, Aiko. Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data. Empirical Software Engineering (2014). DOI 10.1007/s10664-013-9250-3

[3] Sousa, Leonardo da Silva. 2016. Spotting design problems with smell agglomerations. In Proceedings of the 38th International Conference on Software Engineering Companion (ICSE '16). Association for Computing Machinery, New York, NY, USA, 863–866. DOI:https://doi.org/10.1145/2889160.2889273

[4] Khomh, Foutse; Vaucher, Stephane; Guéhéneuc, Yann-Gaël; Sahraoui, Houari. BDTEX: A GQM-based Bayesian approach for the detection of antipatterns, Journal of Systems and Software, 2011, https://doi.org/10.1016/j.jss.2010.11.921.

[5] Cortellessa, Vittorio; Di Marco, Antinisca; Trubiani, Catia. An approach for modeling and detecting software performance antipatterns based on first-order logics. Software and Systems Modelling (2014). DOI 10.1007/s10270-012-0246-z

[6] Khan, Y.A., El-Attar, M. Using model transformation to refactor use case models based on antipatterns. Information Systems Frontiers 18, 171–204 (2016). https://doi.org/10.1007/s10796-014-9528-z

[7] Sharma, Tushar; Fragkoulis, Marios; Spinellis, Diomidis. 2016. Does your configuration code smell? In Proceedings of the 13th International Conference on Mining Software Repositories (MSR '16). Association for Computing Machinery, New York, NY, USA, 189–200. DOI:https://doi.org/10.1145/2901739.2901761

[8] Arnaoudova, Venera; Di Penta, Massimiliano; Antonio, Giuliano; Gueheneuc, Yann-Gael. 2013. A New Family of Software Anti-patterns: Linguistic Anti-patterns. In Proceedings of the 2013 17th European Conference on Software Maintenance and Reengineering (CSMR '13). IEEE Computer Society, USA, 187–196. DOI:https://doi.org/10.1109/CSMR.2013.28

[9] Jaafar, Fehmi; Gueheneuc, Yann-Gael; Hamel, Sylvie; Khomh, Foutse. Mining the relationship between anti-patterns dependencies and fault-proneness. In: 2013 20th Working Conference on Reverse Engineering (WCRE), Koblenz, Germany, 2013 pp. 351-360. doi: 10.1109/WCRE.2013.6671310

[10] Bavota, Gabriele; Qusef, Abdallah; Oliveto, Rocco; De Lucia, Andrea; Binkley, David. An empirical analysis of the distribution of unit test smells and their impact on software maintenance. 2012 28th IEEE International Conference on Software Maintenance (ICSM), 2012, pp. 56-65, doi: 10.1109/ICSM.2012.6405253.

[11] Spadini, Davide; Palomba, Fabio; Zaidman, Andy; Bruntink, Magiel; Bacchelli, Alberto. On the Relation of Test Smells to Software Code Quality. 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 1-12, doi: 10.1109/ICSME.2018.00010

[12] Spadini, Davide; Schvarcbacher, Martin; Oprescu, Ana-Maria; Bruntink, Magiel; Bacchelli, Alberto. 2020. Investigating Severity Thresholds for Test Smells. In 17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3379597.3387453

[13] Abad, Zahra Shakeri Hossein; Karimpour, Reza; HO, Jason; Didar-Al-Alam, S.M.; Ruhe, Guenther; TSE, Edward; Barabash, Kevin; Hargreaves, Ian. Understanding the Impact of Technical Debt in Coding and Testing: An Exploratory Case Study. 2016 3rd International Workshop on Software Engineering Research and Industrial Practice.

[14] Campos, Denivan; Rocha, Larissa; Machado, Ivan. Developers' perception on the severity of test smells: an empirical study. 2021. In: XXIV Ibero-American Congress on Software Engineering (Virtual Event)

[15] Aljedaani, Wajdi; Peruma, Anthony; Aljohani, Ahmed; Alotaibi, Mazen; Mkaouer, Mohamed Wiem; Ouni, Ali; Newman, Christian D.; Ghallab, Abdullatif; Ludi, Stephanie. 2021. Test Smell Detection Tools: A Systematic Mapping Study. In Evaluation and Assessment in Software Engineering (EASE 2021) June 21–23, 2021, Trondheim, Norway. ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3463274.3463335

[16] van Bladel, Brent; Demeyer, Serge. 2018. Test behaviour detection as a test refactoring safety. In Proceedings of the 2nd International Workshop on Refactoring (IWoR 2018). Association for Computing Machinery, New York, NY, USA, 22–25. DOI:https://doi.org/10.1145/3242163.3242168

[17] Bleser, Jonas; Di Nucci, Dario; Roover, Coen. Assessing Diffusion and Perception of Test Smells in Scala Projects. 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).

[18] Alomar, Eman Abdullah; Peruma, Anthony; Mkaouer, Mohamed Wiem; Newman, Christian; Ouni, Ali; Kessentini, Marouane. How we refactor and how we document it? On the use of supervised machine learning algorithms to classify refactoring documentation. Expert Systems with Applications, volume 167, 2021, https://doi.org/10.1016/j.eswa.2020.114176.

[19] Lambiase, Stefano; Cupito, Andrea; Pecorelli, Fabiano; De Lucia, Andrea; Palomba, Fabio. 2020. Just-In-Time Test Smell Detection and Refactoring: The DARTS Project. In Proceedings of the 28th International Conference on Program Comprehension (ICPC '20). Association for Computing Machinery, New York, NY, USA, 441–445. DOI:https://doi.org/10.1145/3387904.3389296

[20] Palomba, Fabio; Zaidman, Andy; De Lucia, Andrea. Automatic Test Smell Detection Using Information Retrieval Techniques. 2018. IEEE International Conference on Software Maintenance and Evolution (ICSME), 2018, pp. 311-322, doi: 10.1109/ICSME.2018.00040.

[21] Dragan, Natalia; Collard, Michael L.; Maletic, Jonathan I. Reverse Engineering Method Stereotypes. 2006 22nd IEEE International Conference on Software Maintenance, 2006, pp. 24-34, doi: 10.1109/ICSM.2006.54.

[22] Wohlin, C.; Runeson, P.; Host, M.; Ohlsson, M. C.; Regnell, B.; Wesslen. A. Experimentation in software engineering. Springer Science & Business Media, 2012

[23] Buse, Raymond P.L.; Weimer, Westley R. 2008. A metric for software readability. In Proceedings of the 2008 international symposium on Software testing and analysis (ISSTA '08). Association for Computing Machinery, New York, NY, USA, 121–130. DOI:https://doi.org/10.1145/1390630.1390647

[24] Posnett, Daryl; Hindle, Abram; Devanbu, Premkumar. 2011. A simpler model of software readability. In Proceedings of the 8th Working Conference on Mining Software Repositories (MSR '11). Association for Computing Machinery, New York, NY, USA, 73–82. DOI:https://doi.org/10.1145/1985441.1985454

[25] Kaur, A. A Systematic Literature Review on Empirical Analysis of the Relationship Between Code Smells and Software Quality Attributes. Arch Computat Methods Eng 27, 1267–1296 (2020). https://doi.org/10.1007/s11831-019-09348-6