Are Code smells the root cause of faults? A continuous experimentation approach

Luca Pellegrini
Free University of Bozen-Bolzano, Italy
luca.pellegrini@stu-inf.unibz.it

Valentina Lenarduzzi
Tampere University of Technology, Finland
valentina.lenarduzzi@tut.fi

ABSTRACT

Code Smells are quite a good instrument to evaluate code's quality, even if they provide a general analysis: no one can assure that determined code smells are really responsible for faults. More and more software companies pay attention to produce qualitative software, in order to reduce the number of bugs. But how can they know, which code-refactoring really can decrease the faults' number? In this work, we aim to find out which code smells are really the cause of bugs and, through a continuous monitoring system, continuously propose companies code refactoring, with the aim of reduce drastically the number of bugs.

KEYWORDS

Code Smells, Fault, Continuous Experimentation

ACM Reference Format:
Luca Pellegrini and Valentina Lenarduzzi. 2018. Are Code smells the root cause of faults? A continuous experimentation approach. In *XP '18 Companion: XP '18 Companion, May 21–25, 2018, Porto, Portugal.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3234152.3234153

1 INTRODUCTION

Continuous improvement is part of several agile practices [1], [2]. Companies keep on improving their process based on different point of view [3]. However, in several cases, developers face the problem of understanding how to improve their code quality [4], [5], [6] so as to reduce corrective maintenance effort [1], [2]. Technical Debt is considered by several practitioners as cause of several faults [7]. However, beside some studies correlated the overall technical debt with faults, as bests, no studies tried to understand which component of the technical debt can be the root cause of software faults.

SonarQube, one of the most common open source tool for software quality analysis proposes an algorithm to calculate the technical debt based on the time needed to refactor code containing coding style violations. Practitioners commonly rely on the Technical Debt definition provided by SonarQube, other customize the syntactical rules removing some of them mainly based on got feeling [8]. SonarQube calculate the Technical Debt based on the effort needed to remove code violations. However, it classifies as "code smells" coding style violations instead of the standard code smells [13]. Other works extended SonarQube considering also the common Code Smells [13],[9] as part of the Technical Debt [10].

In this work we propose a continuous monitoring approach extending [10]. We aim at helping practitioners in understanding which are the code smells or the coding style code violations that commonly generated faults.

The goal of this work is to propose a continuously quality monitoring approach integrated in the DevOps tool-chain to continuously analyze the *bug-inducing-commits* so as to prove/deny that they contain more code smells than other commits and find out which code smells mostly generate bugs.

Based this goal, we derived the following research questions (RQs):
RQ1: Are Code smells the root cause of faults?
RQ2: Which code smells is the root causes of more faults?

In this work we would like to analyze different open source projects, in order to deny or confirm our research questions. Moreover, since several companies are re-architecting their systems based on microservices [23][7], we plan to analyze project developed with monolithic and microservices architectures.

2 BACKGROUND

Code smells were proposed for the first time by Riel [11] and Brown et al. [12]. Then, Beck and Fowler [13] defined a comprehensive set of 22 code smells as harmful anti-patterns that should be removed. Several studies investigated the harmfulness of code smells. However, at bests, all the studies considered fault proneness or change proneness taking into account the frequency of changes in case of presence of code smells.

Inspecting the source code, code smells are significant indicators of a design flaw or problem that can affect negatively the maintainability process [15] increasing the number of faults. Several study investigated the correlation between one or more code smells and the presence of faults in the source code [16], [17], [18], [19] also identifying which code smells were changed more frequently and contained more defects than other classes [19] or which one indicates increased or reduced faultiness [20].

In this work we consider a "fault" as "an incorrect step, process, or data definition in a computer program" as proposed by the IEEE standard 610.12-1990 [14].

3 THE APPROACH

The approach will consider only the data coming from the DevOps tool chain of a single company. This work will be integrated in the continuous building pipeline, and executed every time the build is scheduled. For each project developed by the company, we

XP '18 Companion, May 21–25, 2018, Porto, Portugal

ACM ISBN 978-1-4503-6422-5/18/05…$15.00
https://doi.org/10.1145/3234152.3234153

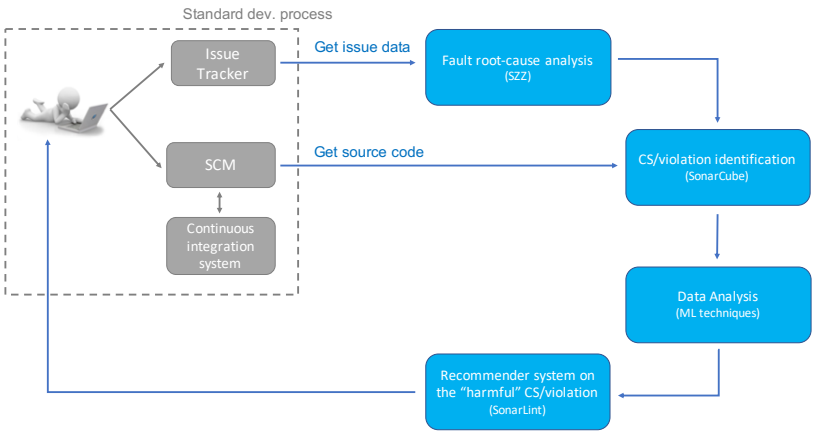

Figure 1: The Approach

will continuously perform the following four steps as depicted in Figure 1:

Step 1 *Analyze and retrieve the fault related commits:* In this step, we will first analyze the log of each commit, so as to understand which commits were related to bug-fixing, or on other tasks. Then, thanks to the SZZ-algorithm [21] we retrieve for each bug-fixing-commit the corresponding bug-inducing-commit(s) (a bug can be also caused from multiple commits). The SZZ algorithm takes as input a source code management (SCM) in a log form and an issues tracker repository (for instance BugZilla / Jira). The algorithm first of all performs a mapping between fixing commits and issues through a syntactic and semantic analysis of the commits classified as faults. Secondly, analyzing files changed of the bug fixing commit and considering the issue's open and close date, it determines the bug-inducing-commit(s). Therefore, the output of this phase is a list of *bug-fixing-commits* and *bug-inducing-commits*.

Step 2 *Identify code smells and code violations:* The main goal of this phase is to analyze code smells and violations (but also the overall *technical depth*) of the last commits. This tasks will be performed by means of SonarQube with the support of the Code Smells Plugin [1]. The output of this step is a list of code smells and sonarqube-violations grouped by commit.

Step 3 *Data Analysis:* In this last step, we want to analyses and combine the results obtained in the first two steps. We first aim to prove or deny that *bug-inducing-commits* contain more code smells than other commits. Our approach here is to compare the number of code smells of different code smells in the *bug-inducing-commits* again those present in the other commits. Secondly, we want to analyze all code smells present in *bug-inducing-commits*, in order to determine, which code smells are the main responsible for bugs for the analyzed project. This step will be performed with a combination of different statistical techniques, including regressions and machine learning algorithms. The analysis will be carried out at a project level. However, we will consider to perform the analysis at a developer level. Moreover, we will also consider the possibility of analyzing the propagation of smells between teams of developers, following an approach similar to [22].

Step 4 *Report developers on the "harmful" code smells and violations by means of a set of IDE plug-ins:* In this step we aim at reporting the results of the analysis to the developers. This task will be carried out thanks to SonarLint [2], a set of SonarQube plugins for the most common IDEs that allows to provide a custom set of coding rules to developers. The set of rules will be dynamically updated every time a new build trigger this process.

4 CURRENT STATUS

In order to validate our approach, we selected a dataset of 23 java projects from the Apache Software Foundation. We already executed SonarQube in all the commits since the beginning of the projects, analyzing more than 1.5 millions of commits.

We already implemented a first version of the SZZ algorithm for extracting data from BugZilla. However, since all the projects in our dataset use Jira as issue tracker we are adapting the SZZ implementation to Jira. Moreover, the actual implementation needs to be improved, in order to be more reliable and precise and its precision and recall needs to be calculated.

Once we will have validated the results, we aim at applying the process in local companies. We do not expect important daily changes in the set of rules, since the updates performed every build will influence only marginally the statistics. In the long run, we could expect to have big changes in rules since after the application of the approach developers will start learning how to not create them and that smell or violation should be reduce its harmfulness.

REFERENCES

[1] D. Taibi, V. Lenarduzzi, P. Diebold, and I. Lunesu. Operationalizing the Experience Factory for Effort Estimation in Agile Processes. International Conference on Evaluation and Assessment in Software Engineering. pp. 31-40. (2017)
[2] V. Lenarduzzi, I. Lunesu, M. Matta and D. Taibi. Functional Size Measures and Effort Estimation in Agile Development: A Replicated Study. Agile Processes in Software Engineering and Extreme Programming. pp. 105-116. (2015)
[3] D. Taibi and V. Lenarduzzi. MVP explained: A Systematic Mapping on the Definition of Minimum Viable Product. Euromicro Conference on Software Engineering and Advanced Applications (SEAA2016). pp. 112-119. (2016)

[1] SonarQube Anti-Patterns Code Smells Plug-in - https://github.com/davidetaibi/sonarqube-anti-patterns-code-smells

[2] Sonarlint - https://www.sonarlint.org/

[4] D. Tosi, L. Lavazza, S. Morasca, and D. Taibi. On the definition of dynamic software measures. International symposium on Empirical software engineering and measurement. pp. 39-48. (2012)

[5] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi. Predicting OSS trustworthiness on the basis of elementary code assessment. International Symposium on Empirical Software Engineering and Measurement. Art.36, 4 pages. (2010)

[6] L. Lavazza, S. Morasca, D. Taibi, and D. Tosi. An empirical investigation of perceived reliability of open source Java programs. Annual ACM Symposium on Applied Computing. pp.1109-1114. (2012)

[7] D.Taibi, V. Lenarduzzi and C. Pahl. Processes, Motivations and Issues for Migrating to Microservices Architectures: An Empirical Investigation. IEEE Cloud Computing. Vol. 4(5), pp. 22-32. (2017)

[8] D. Taibi, A. Janes and V. Lenarduzzi. How developers perceive smells in source code: A replicated study. Information and Software Technology, vol. 92, pp. 223-235. (2017)

[9] D. Taibi and V. Lenarduzzi. On the Definition of Microservice Bad Smells. IEEE software. Vol 35, Issue 3, May/June (2018)

[10] V. Lenarduzzi, A. C. Stan, D. Taibi, D. Tosi and G. Venters. A Dynamical Quality Model to Continuously Monitor Software Maintenance. European Conference on Information Systems Management. pp. 168-178. (2017)

[11] A. J. Riel. Object-oriented design heuristics. Addison-Wesley Reading. Vol. 335. (1996)

[12] W. H. Brown, R. C. Malveau, H. McCormick and T. J. Mowbray. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley and Sons, Inc. (1998)

[13] M. Fowler, K. Beck and J. Brant. Refactoring: Improving the Design of Existing Code. Addison-Wesley Longman Publishing Co., Inc. (1999)

[14] IEEE Standard Glossary of Software Engineering Terminology. standards coordinating committee of the Computer Society of the IEEE. IEEE Std 610.12-1990

[15] S. L. Natthawute, H. Shinpei, S. Motoshi. Context-based code smells prioritization for prefactoring. International Conference on Program Comprehension. (2016)

[16] F. Rahman, C. Bird, and P. Devanbu. Clones: What is that smell?. Empirical Software Engineering, 17(4-5), pp.503-530. (2012)

[17] E. Juergens, F. Deissenboeck, B. Hummel and S. Wagner, S. May. Do code clones matter?. International Conference on Software Engineering (ICSE 2009), pp. 485-495. (2009)

[18] W. Li and R. Shatnawi. An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. Journal of systems and software, 80(7), pp.1120-1128. (2007)

[19] S.M. Olbrich, D.S. Cruzes and D.I. SjÃÿberg. Are all code smells harmful? A study of God Classes and Brain Classes in the evolution of three open source systems. IEEE International Conference on Software Maintenance, pp. 1- 10. (2010)

[20] T. Hall, M. Zhang, D. Bowes and Y. Sun. Some code smells have a significant but small effect on faults. ACM Transactions on Software Engineering and Methodology, vol. 23(4), p.33. (2014)

[21] J. Sliwerski, T. Zimmermann, A. Zeller. When Do Changes Induce Fixes? International workshop on Mining software repositories. pp.1-5. (2005)

[22] B. Carminati, E. Ferrari, S. Morasca, and D. Taibi. A probability-based approach to modeling the risk of unauthorized propagation of information in on-line social networks. First ACM conference on Data and application security and privacy (CODASPY '11). pp. 51-62. (2011)

[23] D. Taibi, V. Lenarduzzi, C. Pahl, and A. Janes. Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. XP2017 Scientific Workshops (XP '17). Article 23, 5 pages. (2017)