2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)

Towards Just-In-Time Rational Refactoring

Jevgenija Pantiuchina
jevgenija.pantiuchina@usi.ch
REVEAL @ Software Institute, *Università della Svizzera italiana (USI)*, Lugano, Switzerland

***Abstract*—Code smells have been defined as symptoms of poor design and implementation choices. Empirical studies showed that code smells can have a negative impact on the maintainability of code. For this reason, tools have been developed to automatically detect design flaws and, in some cases, to recommend developers how to remove them via refactoring. However, these tools are not able to *prevent* the introduction of design flaws. This means that the code has to experience a quality decay before state-of-the-art tools can be applied. In addition, existing tools recommend refactoring operations that mostly target the improvement of quality metrics (*e.g.,* cohesion) rather than the generation of refactorings that are meaningful from the developers' perspective. Our goal is to develop techniques serving as the basis for a new generation of refactoring recommenders able to (i) predict code components likely to be affected by code smells in the near future, to refactor them before they experience a quality decay and (ii) recommend meaningful refactorings emulating the ones that developers would perform, rather than the ones targeting the improvement of metrics. We refer to such a perspective on refactoring as just-in-time rational refactoring.**

***Index Terms*—Code quality metrics, maintenance, refactoring**

I. Introduction

Empirical studies have shown that software complexity tends to increase over time due to continuous maintenance and evolution activities. To tackle complexity and maintenance, high software quality must be ensured. Design flaws, anti-patterns, and code smells are terms used in the literature to indicate poor design choices made by developers during coding activities, which may trigger refactoring operations aimed at removing them [3]. To detect design flaws and recommend possible refactorings, tools and techniques have been developed in industry and academia. For example, Marinescu [5], in his work on design flaws detection, uses static code analysis to compute the "detection strategies" – metric-based rules that capture deviations from good design principles. Detection based on static code analysis has been then used by many other researchers [6] as well as in tools (PMD, iPasma [4]). Other authors used change-history information mined from the versioning systems to identify specific types of design flaws [7]. Finally, Palomba *et al.* [8] proposed a technique exploiting textual analysis to detect code smells.

While there is a wide variety of techniques to identify design flaws, none of them is able to *prevent* the introduction of such design flaws, neither to recommend refactorings emulating the ones that developers would perform, rather than the ones targeting the improvement of quality metrics. To address these limitations we aim to develop techniques serving as the basis for a new generation of refactoring recommenders.

II. Current Status of Research And Future Plans

Towards our objective we defined two research goals.

Goal 1. Predicting Code Quality Decay. We aim to develop techniques able to alert the developer when a code component is deviating from good design principles, before design flaws are introduced. As a first step, we have proposed an approach named COSP (**Co**de **S**mell **P**redictor) [9], that is able to predict whether a given class will be affected by a specific type of code smell within t days. In our preliminary work, we used COSP to predict classes that will become *God* and *Complex Classes* within $t = 90$ days. We trained the machine learner using the Weka implementation of the Random Forest [2] with the goal of identifying a series of rules which would discriminate classes likely to be affected by a specific smell type within t days. As predictor variables we used 42 metrics capturing object-oriented quality aspects of a class C including complexity, cohesion, coupling, data hiding, and inheritance. Random Forest classifies previously unseen classes providing as output the probability that a class belongs to each of the possible categories of the dependent variable *i.e.,* will become smelly or not. We exploit this indication as a *confidence level* for COSP. In particular, given a scenario in which the developer uses COSP to prioritize classes for refactoring, she can start by only inspecting the ones predicted by COSP as becoming smelly with a probability of 90%.

The results show that with the maximum confidence level (1.0), COSP discriminates classes likely to be affected by *God Class* or *Complex class* smell types within 90 days with a precision of $\sim$75%, but with recall not higher than $\sim$13% [9]. COSP represents the very first solution paving the way to more research in predicting classes that will hinder code maintainability. We are working on improving the prediction accuracy of COSP investigating the usage of other types of predictor variables, such as process metrics (*e.g.,* the class change- and fault-proneness, the developers experience).

Given the ability to identify code components that are likely to become maintainability issues in the future, the next step is to recommend refactoring operations to developers. The existing state-of-the-art refactoring recommenders exploit quality metrics as "fitness function" with the goal of maximizing code quality as assessed by these metrics. Thus, they imply a strong assumption: Quality metrics are able to assess code quality as perceived by developers. Indeed, refactoring recommenders should be able to suggest refactorings that are meaningful from the developers' point-of-view.

2574-1934/19/$31.00 ©2019 IEEE
DOI 10.1109/ICSE-Companion.2019.00073

While such an assumption might look reasonable, there is limited empirical evidence supporting it, mostly based on studies that investigated this phenomenon by surveying developers [1], [11]. To validate this assumption we analyzed real changes (commits) implemented by developers with the stated purpose of improving a specific quality attribute. Then, we used seven state-of-the-art quality metrics to assess the change brought by each of those commits to the specific quality attribute it targets. The goal was to investigate whether the quality improvement expected by the developer is also reflected in the metrics' values. Our results show that, more often than not, the studied quality metrics are not able to capture the quality improvements as perceived by the developers [10].

Our findings highlight the possible limitations of software engineering applications built on top of quality metrics (*e.g.,* recommender systems aimed at identifying code smells and suggest refactorings). Indeed, while a refactoring could make sense from the quality metrics' point of view, it might be meaningless for developers (or *vice versa*). This finding motivates our second research goal: Learn refactoring operations from developer' activities, rather than recommending refactorings maximizing some quality metrics.

Goal 2. Learning Refactoring Transformations. Our goal is to investigate the possibility to apply deep learning models to learn code changes performed by software developers. We will use an approach similar to the one proposed by Tufano *et al.* [14] to automatically fix bugs. The authors used Neural Machine Translation (NMT) models to automatically "translate" buggy code into fixed code. They mined bug-fixing commits from GitHub and extracted method-level AST edit operations applied by developers to the buggy code to implement the patch. This process resulted in pairs of code components (before and after the fix). Using this data, the authors trained an Encoder-Decoder Recurrent Neural Network to learn the code transformations performed by developers during bug fixing activities. We applied the same approach investigating whether a NMT model can replicate the code transformations performed by the developers in the pull requests [13].

In future, we plan to narrow the scope of code transformations to only refactorings with the goal to investigate whether a NMT model can replicate refactoring operations performed by software developers. This can be accomplished by mining a large set of refactoring operations using existing tools [12], and using this data to train the NMT model on pairs of code components before and after the application of refactoring.

III. Summary

Towards our envisioned just-in-time rational refactoring recommender, we started by developing COSP (COdel Smell Predictor)—an approach able to predict code components that will represent maintainability issues in the near future. Our first prototype can identify problematic components with a precision of $\sim$75%, but still suffers of low recall ($\sim$13%), thus requiring more research. For the envisioned refactoring recommender, we studied whether quality metrics can capture improvements in code quality as perceived by developers.

We found that, in most of the cases, there is no alignment between the quality improvement as perceived by developers and how assessed by metrics. This justifies the need for exploring novel refactoring recommenders. Thus, we plan to employ a NMT model to learn code transformations performed by developers in commits implementing refactoring operations. Starting from a previously unseen code component given as input, the NMT model should be able to replicate refactoring operations as implemented by developers.

Acknowledgment

We gratefully acknowledge the Swiss National Science Foundation for the JITRA project (No. 172479).

References

[1] G. Bavota, B. Dit, R. Oliveto, M. D. Penta, D. Poshyvanyk, and A. D. Lucia. An empirical study on the developers' perception of software coupling. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*, pages 692–701.

[2] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[3] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co. Inc., Boston, MA, USA, 1999.

[4] C. Marinescu, R. Marinescu, P. F. Mihancea, D. Ratiu, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *ICSM (Industrial and Tool Volume)*, pages 77–80, 2005.

[5] R. Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, ICSM '04, pages 350–359, Washington, DC, USA, 2004. IEEE Computer Society.

[6] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur. Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.*, 36(1):20–36, Jan. 2010.

[7] F. Palomba, G. Bavota, M. D. Penta, R. Oliveto, A. D. Lucia, and D. Poshyvanyk. Detecting bad smells in source code using change history information. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 268–278, Nov 2013.

[8] F. Palomba, A. Panichella, A. D. Lucia, R. Oliveto, and A. Zaidman. A textual-based technique for smell detection. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, pages 1–10, May 2016.

[9] J. Pantiuchina, G. Bavota, M. Tufano, and D. Poshyvanyk. Towards just-in-time refactoring recommenders. In *Proceedings of the 26th Conference on Program Comprehension, ICPC 2018, Gothenburg, Sweden, May 27-28, 2018*, pages 312–315, 2018.

[10] J. Pantiuchina, M. Lanza, and G. Bavota. Improving code: The (mis)perception of quality metrics. In *34th IEEE International Conference on Software Maintenance and Evolution (ICSME 2018) Madrid, Spain, May 23-29, 2018*, 2018.

[11] M. Revelle, M. Gethers, and D. Poshyvanyk. Using structural and textual information to capture feature coupling in object-oriented software. *Empirical Software Engineering*, pages 773–811, 2011.

[12] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 483–494, New York, NY, USA, 2018. ACM.

[13] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. In *Proceedings of the 41th International Conference on Software Engineering*, ICSE '19. IEEE, 2019.

[14] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical investigation into learning bug-fixing patches in the wild via neural machine translation. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 832–837, New York, NY, USA, 2018. ACM.