When, why and for whom do practitioners detect technical debt?: An experience report

Norihiro Yoshida
Nagoya University
Email: yoshida@ertl.jp

***Abstract*—Code cloning is one of the most well-known code-level technical debts. In this paper, I discuss when, why and for whom practitioners detect code clones based on my experience of industry/university collaboration. At first, I introduce five project instances based on my experience. Next, I identify elements of the context model of a software maintenance project. After that, I discuss the impact of the context of a software maintenance project on technical debt.**

I. Introduction

The results of empirical studies of software engineering usually depend on the context (e.g., programming language. size of product, development process) of a software development [1]. Studies of technical debt [2] also depend on it. For example, if a development team expects to perform long-term software maintenance, it proactively refactors source code. Conversely, if it is commissioned to maintain the source code for a customer and do not expect to touch it after the project, it is unmotivated to perform refactoring.

So far, much research has been done on the empirical studies of technical debt [3], [4], [5]. The results of those studies sometimes tell different stories. For instance, several studies successfully detected defects that are caused by maintaining code clones [6], [7]. On the other hand, other studies reported that code clones are harmless for software maintenance [8], [9]. Such inconsistency can be caused by context differences between target projects in the empirical studies of technical debt. For a deeper understanding of technical debt, the empirical software engineering community has to identify the context model of a software maintenance project.

In this position paper, I focus on code cloning which is one of the most well-known examples of code-level technical debt [6], [7], [10]. At first, I introduce the context instances when developers focus on code cloning according to my experiences in industry/university collaboration. After that, I discuss elements of the context model of a software maintenance project and the impact of those elements on technical debt.

The remainder of this position paper is organized into the following sections. Section II introduces the context instances when developers focus on code cloning according to my experiences in industry/university collaboration. Next, Section III discusses elements of the context model of a software maintenance project and the impact of those elements on technical debt. Section IV reviews related work and finally, Section V concludes with possible future work.

II. Project Instances

Case A: The company of Case A is a Japanese conglomerate company. A division of this company has owned and maintained two large-scale legacy systems written in the C language. One of the systems is for an electric power company, and the another one is for a railway company. The developers in the division expect to maintain the code in the next decade. Since they plan to perform large-scale maintenance soon, they would like to reduce the amount of the code by merging code clones. They believe that they will be able to reduce the cost of maintaining the code once clones in the code are merged.

Case B: An another division of the company in Case A provides service for reducing legacy code written in COBOL. So far, the software system often has dependencies on a specific vendor for products and services and the customer of the system has been unable to use another vendor without substantial switching cost. Recently, many customers would like to migrate from such a vendor lock-in system to a new system that uses open source software (e.g., Linux, PostgreSQL). Before the migration, the customers would like to reduce the existing source code and reduce the maintenance cost.

Case C: The company of Case C is also a conglomerate company. This company has owned and maintained large-scale software systems for smartphones. The code is written in the C language. The amount of code has rapidly increased recently. The persons in charge worry about inconsistencies among code clones and would like to perform simultaneous modifications correctly.

Case D: The company of Case D is a Japanese system integration company. In Case D, several subcontractors contribute to the project, and each of them has been in charge of a part of the development. Each subcontractor has maintained subcontractor-owned code for the part and delivered it after implementation phase. The project manager in case D would like to know the location of code clones in large-scale source code and avoid the amount increases.

Case E: The company of Case E is a Japanese provider of information technology services and products. Case E is a maintenance project for medium-scale source code that is owned by this company. The developers in Case E would like to avoid that the amount of code clones increases and detect newly-created or modified clones on the fly. They would like to use a system for reporting such clones daily.

Table I summarizes the characteristics of above project instances in terms of not only when, why and for whom practitioners detect clones but also language and owner/scale of a code base.

III. Discussion

A. Context Element

According to Table I in Section 2, we found that the context model of software maintenance projects includes the following elements for the empirical software engineering of technical debt:

- When is technical debt detected? (e.g., before releasing a version, daily build)
- Why are stakeholders motivated to detect technical debt? (e.g., refactoring, clone prevention)
- Who is expected to check detected debt? (e.g., development team, customer who would like to maintain the system, project manager)
- Who is the owner of the code? (e.g., company that developers who detect technical debt own, customer who would like to maintain the system)
- What kind of the code? (e.g., language, legacy/new, long-term/short-term maintained)

B. Impact of Context on Technical debt

Hereafter, I discuss the impact of the context elements in Section III-A on technical debt.

a) When: In order to detect technical debt as much as possible, it is most appropriate to monitor all of the code modifications on the fly because many refactoring operations are expected to be completed before a commit. Second best is detecting technical debt from all of the committed versions in a version control system. A released version is expected to include a fewer number of technical debt compared to a committed version because developers not only tend to introduce technical debt but also try to reduce them. *When researchers compare or discuss empirical studies of technical debt, they have to be careful about the timing when technical debt are detected.*

b) Why: The result of an empirical study of technical debt is expected to depend on the strategy to deal with it in the development. If developers considered refactoring as a solution to code clones, the number of the code clones tends to be small. Conversely, if developers considered to keep the consistency among code clones and left consistent clones as they are, the number of code clones is larger than the previous case. *Researchers should try to find out the strategy to deal with technical debt in the development and the purpose of it.*

c) For Whom and Owner: In the case that technical debt was detected in company-owned code by a maintenance team, it may have carefully considered a strategy to deal with each of the technical debts for maintenance in the future. In this case, many of the detected technical debts are expected to be eliminated successfully, and the maintenance cost of the code is also expected to reduce. Conversely, in the case that technical debt was detected in customer-owned code for a customer, the customer tends to focus on the amount of technical debt and does not consider a strategy to deal with each of them. The case that technical debt was detected in subcontractor-owned code by a project manager is very similar to the above case. The project manager tends just to focus on the amount of technical debt and does not consider a strategy to deal with each of them. In these cases, the decrease of the maintenance cost is expected to be limited even if many of the technical debt are eliminated. *Researchers have to be careful about persons in charge of checking detected technical debt and the owner of the code when they compare or discuss empirical studies of the technical debt.*

d) Target: The expressiveness of a programming language strongly affects the strategy to deal with code smells [11], [12]. For example, Java has many language features for refactoring, but C has only a few such ones. Therefore, *researchers have to be careful about a programming language that is used in the project when they compare or discuss empirical studies of the technical debt.* The size of a code base also is considered to affect the number of code smells. Keeping the consistency of large-scale source code is difficult. For example, large-scale source code tends to include many code clones and it is difficult to keep the consistency of the code clones [6], [13].

IV. Related Work

Defect-prone clone is regarded as a serious technical debt. Many empirical studies have focused on the relationship between code cloning and defects. Several researchers investigated the relationship between code clones and defects in source code. Rahman et al. reported that the great majority of defects are not significantly associated with code clones [8]. Also, Sajnani et al. reported that code clone has considerably less, and less problematic, bug patterns [9]. Mondal et al. compared the defect-proneness of different type of code clones [14]. Islam et al. reported that a considerable proportion of the code clones was able to contain replicated bugs [15].

The inconsistency among code clones is a clue to the detection of defect-prone clones. Li et al. proposed an a tool, CP-Miner that uses data mining techniques to efficiently identify copy-pasted code in large software suites and detects copy-paste defects based on naming inconsistency among code clones [16]. Jiang et al. proposed an approach to detecting clone-related defects based on inconsistencies among clones [6]. Juergens presented the results of a large-scale case study that was undertaken to find out if inconsistent changes to cloned code can indicate defects [13]. They not only found that inconsistent changes to clones are very frequent but also identified a significant number of defects induced by such changes.

Change-prone code is a clue to the detection of technical debt. Several researchers investigated the change-proneness of code clones. Hotta et al. reported that the presence of duplicate code does not have a negative impact on software evolution [17]. Harder and Göde reported that clone stability varies

TABLE I
COMPARISON OF PROJECT INSTANCES

Case	When	Why	for Whom	Owner	Target
A	before large-scale maintenance	refactoring	maintenance team	company	large-scale legacy C code
B	before legacy modernization	refactoring	customer	customer	large-scale legacy C/COBOL code
C	during maintenance	consistency management	maintenance team	company	large-scale legacy C code
D	after implementation phase	prevention	project manager	subcontractor	large-scale legacy C/C+ code
E	everyday during maintenance	prevention	maintenance team	company	medium-scale legacy Java code

depending on the clones characteristics, the corresponding project environment, and over time [18].

Code-level technical debt is not limited to code cloning. Yamashita and Moonen investigated the relationship between code smell and maintainability [3], [4]. They investigated the capability of 12 code smells to reflect actual maintenance problems [3]. They also empirically investigated the interactions amongst 12 code smells and analyze how those interactions relate to maintenance problems [4].

Refactoring is the most well-known technique aiming to reduce code-level technical debt. Several empirical studies have been done on the relationship between code smells and refactoring. Stroggylos and Spinellis investigated the impact of refactoring on quality metrics [19]. Bavota et al. investigated the extent to whether refactorings executed on classes exhibiting code smells and able to remove code smells [20]. The result shows that 42% of refactoring operations were performed on code entities affected by code smells. However, only 7% of the performed operations actually removed the code smells from the affected class. Also, Saika et al. investigated the impact of the severity of code smell on refactoring [21]. The result shows that refactoring did not decrease the severity of code smells significantly.

V. Concluding Remarks and Future Work

In this position paper, I focused on code cloning that is one of the most well-known code-level technical debt At first, I introduced five context instances when developers focus on code clones according to my experience of industry/university collaboration. After that, I discussed elements of the context model of a software maintenance project and the impact of those elements on technical debt. Based on the discussion, my position statement is that *researchers have to identify the contexts of target projects in empirical studies of technical debt before they compare or discuss those studies.*

As future work, I plan to perform a systematic review of the existing empirical studies of technical debt and then identify the contexts of those studies. After that, I would like to categorize the identified contexts and then investigate the impact of context on technical debt. Also, I would like to propose a guideline for empirical research of technical debt based on above category and the investigation result.

Acknowledgment

I thank Dr. Leon Moonen and Prof. Tom Mens for useful feedback on earlier versions of this paper. This work was supported by JSPS KAKENHI Grant Numbers JP26730036 and JP16K16034.

References

[1] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg, “Preliminary guidelines for empirical research in software engineering,” *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 721–734, Aug 2002.

[2] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Software*, vol. 29, no. 6, pp. 18–21, Nov 2012.

[3] A. Yamashita, “Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data,” *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 2013.

[4] A. Yamashita and L. Moonen., “Exploring the impact of inter-smell relations on software maintainability: An empirical study,” in *Proc. of ICSE*, 2013, pp. 682–691.

[5] M. Tufano, F. Palomba, G. Bavota, and R. Oliveto, “When and why your code smell starts to smell bad,” in *Proc. of ICSE*, 2015, pp. 403–414.

[6] L. Jiang, Z. Su, and E. Chiu, “Context-based detection of clone-related bugs,” in *Proc. of ESEC/FSE*, 2007, pp. 55–64.

[7] Y. Higo, Y. Ueda, S. Kusumoto, and K. Inoue, “Simultaneous modification support based on code clone analysis,” in *Proc. of APSEC*, 2007, pp. 262–269.

[8] F. Rahman, C. Bird, and P. Devanbu, “Clones: What is that smell?” *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[9] H. Sajnani, V. Saini, and C. V. Lopes, “A comparative study of bug patterns in java cloned and non-cloned code,” in *Proc. of SCAM*, 2014, pp. 21–30.

[10] N. Yoshida, T. Hattori, and K. Inoue, “Finding similar defects using synonymous identifier retrieval,” in *Proc. of IWSC*, 2010, pp. 49–56.

[11] C. J. Kapser and M. W. Godfrey, ““cloning considered harmful” considered harmful: patterns of cloning in software,” *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9076-6

[12] J. L. Overbey, F. Behrang, and M. Hafiz, “A foundation for refactoring c with macros,” in *Proc. of FSE*, 2014, pp. 75–85. [Online]. Available: http://doi.acm.org/10.1145/2635868.2635908

[13] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter?” in *Proc. of ICSE*, 2009, pp. 485–495. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2009.5070547

[14] M. Mondal, C. K. Roy, and K. A. Schneider, “A comparative study on the bug-proneness of different types of code clones,” in *Proc. of ICSME*, 2015, pp. 91–100. [Online]. Available: http://dx.doi.org/10.1109/ICSM.2015.7332455

[15] J. F. Islam, M. Mondal, and C. K. Roy, “Bug replication in code clones: An empirical study,” in *Proc. of SANER*, vol. 1, March 2016, pp. 68–78.

[16] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code,” *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[17] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution?: An empirical study on open source software,” in *Proc. of IWPSE-EVOL*, 2010, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1862372.1862390

[18] J. Harder and N. Göde, “Cloned code: stable code,” *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1063–1088, 2013. [Online]. Available: http://dx.doi.org/10.1002/smr.1551

[19] K. Stroggylos and D. Spinellis, “Refactoring–does it improve software quality?” in *Proc. of WoSQ*, no. 10, 2007.
[20] G. Bavota, A. D. Lucia, M. D. Penta, and R. Oliveto, “An experimental investigation on the innate relationship between quality and refactoring,” *Journal of Systems and Software*, vol. 107, pp. 1–14, 2015.
[21] T. Saika, E. Choi, N. Yoshida, S. Haruna, and K. Inoue, “Do developers focus on severe code smells?” in *Proc. of PPAP*, 2016, pp. 1–3.