A Flowchart for Rapid Technical Debt Management Decision Making

Congyingzi Zhang*, Yan Wu
Bowling Green State University, Bowling Green, Ohio, U.S.

* Corresponding author. Email: czhang@bgsu.edu, yanwu@bgsu.edu
Manuscript submitted August 31, 2015; accepted December 1, 2015.
doi: 10.17706/jsw.11.2.212-219

Abstract: Technical debt is known as delaying certain software maintenance tasks during software development life cycle to meet development goals in a short run. Such compromise increases maintenance cost in a way like growing financial interest in later development of software life cycle. Despite all the negative impact induced by technical debt, enterprise architects have to leverage between incurring technical debts and meeting short-term customer and financial related goals while making the decision for whether to pay the technical debt. However, the decision making, management, and tracing for technical debt is never formalized and surprisingly done by ad hoc. This study is aimed for constructing a step-by-step decision making map which helps the decision makers consider all possible impact factors. More importantly, such standardized process would improve the efficiency, consistency, stakeholders' communication, and documentation for technical debt handling throughout the software life-cycle. In order to evaluate the effectiveness of the proposed decision making map and the according practice, a survey is composed as the appendix of this research report. This preliminary work, although with resource constraint, is a great foundation to build on for further study.

Key words: Decision making, refactoring, software engineering, technical debt management.

1. Introduction

Widely accepted by the software engineer community, technical debt features sacrificing one dimension of development, for example software quality, to obtain growth on another dimension, such as meeting the deadline with promised product features [1]-[4]. The famous analogy of incurring and paying debt came from Cunningham's report in 1992 on OOPSLA conference. He mentioned that during an incremental development, consolidating the code base by promptly rewriting and revising is the key for the sustainable growth of the product. He coined the process as Incremental Design Repair which they believed is the best way leading them to the most appropriate product in reasonable amount of time. He drew the analogy between shipping first time code and going into debt. A little debt does not jeopardize the project as long as it is paid back promptly by rewriting. However, it would cause the entire project to a stand-still when the accumulated debt costs all the flexibility and consolidation of a product [5]. Technical debt has negative impacts on software products such as low code quality, low maintainability and increased cost in later development. These tradeoffs have to be compromised to satisfy some other dimensions of software product, such as meeting the requirement of new features before the deadline.

As a solution for paying technical debt, researches proposed code refactoring as a practice to alleviate

technical debt problem in an ongoing project. Code refactoring features code transformations while preserving the same external behavior. Through code refactoring, programmers would be able to optimize the code structure and rewrite corrupted code to improve the maintainability and performance of the software product. Fowler and his colleagues [6] brought the term of code smell to this research field referring to certain code structures in that screaming for the possibility of refactoring. They have categorized and discussed the design pattern for several featured code behaviors in the book, such as duplicated code, long method, large classes, parallel inheritance hierarchies, message chains, data classes, and the others. These are good indicators suggesting programmers to refactor their code for avoiding deeper system problems. More importantly in this study, knowing how to refactor code does not mean knowing when to perform it. Also, a large reason that keeps developers from exercising these recommended practices is that addressing technical debt usually means large uncertainty in time and resource investment. To solve such problem, approaches such as automatic code smell detection, domain-specific code smell detection, and other policies were proposed in some prior researches, which were dedicated to find a balancing point of improved code quality and investment of resources. However, the study of technical debt management of enterprise-level products shows that the support from automated code smell detecting tool is very limited and too low-level for addressing the architectural refactoring needs [7], [8]. Although code refactoring was highly recommended for clearing technical debt, the practical meaning of code refactoring in an ongoing project usually deviates from its designed original in various software development scenarios. Kim *et al*. found from a survey conducted with a developers in a large software company that the common narratives of this process are considered costly and risky, which can be a short version of the reason why technical debt exists all the time [7].

The lack of confidence of performing code refactoring comes from an unclear estimation of workload and task complexity. A guideline standardizing decision making on technical debt management is missing. The approaches from current research addressing the technical debt management problem are either too high-level or too costly in man hours. An all-in-one map would be an initial step for formalizing the decision making process tailored to the situation. Many research works in this field discussed the factors upon which the stakeholders, non-technical and technical, decide whether a code refactoring is worth the effort. These factors are the properties of an ongoing project, such as product size, tool support, time constraints, and so on. It is hypothesized that if the most common factors under such circumstance are identified, categorized, and presented on a map indicating their causal relationship, developers could have a better estimation for feasibility of a code refactoring based on the project status. As a living document, the flowchart could be revised to accommodate for a particular development environment and contribute to product development as one of the routine practices because of its low cost, efficiency, easy-following, and effectiveness for handling technical debt during a development life cycle.

The goal and expected outcome of this research is to construct a decision flowchart for developers to follow at any time of a development life-cycle. The diagram would help them identify more decision making factors from their own situation. After that, they would be able to traverse down and find the estimation by following the causal relations among the factors. At the forks of the diagram, they would have a chance to evaluate by the complexity and weight of the considered factors to make a sub-decision until they reach the final estimation. This method is easy-to-follow and not time-consuming, which makes it a simple add-on activity to development daily routine.

2. Literature Review

In order to identify the factors from real world cases that have an impact on the decision making process of technical debt management, the related works and literature review section are organized by discussion

over ten research questions over this topic.

*What is the bad consequence expected from unpaid technical debt?*In Zazworka *et al*.'s research on examining the effect from unfixed technical debt throughout the SDLC, they found technical debt caused growing debt and slowed down development. They suggest technical debt should be closely identified and managed during the development process [3]. In another research study for finding how technical debt affects a project, a delayed maintenance task was tracked throughout the SDLC. After comparing the technical debt cost from the actual project and simulated project, the researchers found out the actual project contains more defects than the simulated [1].

Despite the bad consequences from unpaid technical debt, why it happens enterprise product manager choose trading off long-term cost for short-term gain? There are several factors that could introduce intentionally incurring technical debt into the project. Some are considered less harmful and even necessary when handled properly.

According to an enterprise software management study by Klinger, in real world scenarios, it seems that avoiding technical debt is not considered the first priority as recommended in academic and research areas. Obviously, the technical stakeholders are experienced enough to handle the level of acquiring technical debt in the circumstances when they can foresee paying less effort on code refactoring after the release.

The second source of incurred technical debt is from unexpected changes in requirements, time, or development environment. For example, the unexpected cascaded impact and other unexpected events could be a factor such as having new requirements from the changing market. The technical debt induced under this circumstance can possibly lead to a cost and is difficult to avoid.

The third source of incurred technical debt is from gaps in communication between technical and non-technical stakeholders. In the study [8], they found that the decision maker for code refactoring is usually the non-technical stakeholder instead of the system architect. To improve the global communication, the technical stakeholders have to remind the non-technical stakeholders of paying back the tech debt with a reliable estimation of potential cost and effectiveness of a code rewrite. So, all dimensions of the project will be collectively considered and the management plan would be optimized globally for the people and the product [8]. For example, Narayan *et al*. closely examined the different tradeoff patterns in enterprise software. According to their result, they proposed some actionable policies for technical debt management for large commercial enterprise software product packages [9].

*From the developers' perspective, why they hesitate about refactoring during development?*According to the surveys toward developer's perspective over code refactoring, there are some common fears which hold them from paying technical debt on time.

First of all, the surveyed developers had different understandings of the definition of code refactoring. Recalling its academic definition, code refactoring is not supposed to change code external behavior. However, it is reported from the industry that it sometimes does lead to code behavior changes. It was further investigated and found that rarely does anyone do pure refactoring revisions. In other words, most code refactoring was done when combined with other behavior-changing code modifications [7], [10]. Another common and obvious fear comes from the huge amount of code base and high complexity of inter-component dependency, which is usually observed on legacy system and enterprise-level software package.

Developers reported their confusion after rename and move refactoring because version control system usually is sensitive to such changes. So, without sufficient tool support for merging and integrating refactored code they start losing track of their object in the code history. It was reported that more than half of them do all of their refactoring manually even though they have been aware of the refactoring tool [11].

*Is there any automated tool support for code refactoring?*Post development automatic refactoring includes

using automated architectural level identification of code smell to discover the sites for refactoring, or paying technical debt. For example, a plug-in was designed for Eclipse to detect inappropriate use of inheritance [10]. The limitation of such specifically designed post-fix plugin is obvious because there is no such plug-in for all types of code smell. Even though it might benefit from a fast detection rate, it still requires a huge amount of man hours to review and fix each occurrence of the same type of code smell. Fontana *et al*. research suggests paying the design technical debt before than other types of technical debt [12], [13].

They categorized and found domain-specific code smell and anti-pattern, which might help to focus and alleviate the complexity of code smell detecting algorithm for the automatic architectural level detection [14].

What types of technical debt are 'quick fixes'? What are not? To clarify, 'quick fixes' exist along with the code development and are also known as low-level refactoring. Such refactoring is easier to pay back. To compare with large code bases which require architectural-level refactoring does not considered done with 'quick fixes'. To perform refactoring in a large code base, a top-down scheme could be applied. The other approach requires investment in customized refactoring support tools for post-refactoring quality check [7].

What are the motivations eventually make developers refactor their code? Over half of the code refactoring is done with other code changes. Such changes are made because of development needs. It is less often for a developer to initiate a code rewrite because he or she realized somepotential and uncertain benefits from a code refactoring. The second high motivation for a developer fixing their code is the poor code readability. The third factor make them perform refactoring is fixing duplicated code.

What are the motivations eventually make architects order the performance of an architectural/high-level code refactoring? Usually, several architects initiate the discussion and suggest for a code refactoring upon the observation of large inter-modular dependency and parallel team development. The effort is worthy because they want to maximize parallel development efficiency so the rebuild and retest won't fail when two parallel development teams merge their code. Code reuse for fast release of new product is also considered a reason for lowering the inter-module dependency through high-level code refactoring.

What are the known benefits of code refactoring? If code is refactored during prior development, defects in later development will decrease in number [14]. Code refactoring would help to save cost in code maintenance from repairing potential issues from technical debt [15]. It also helps to avoid new technical debt [16]. To summarize, the benefits from refactoring are considered as improved readability, maintainability, modularity, testability, code performance, ease for adding features, fewer bugs, significant reduction in inter-module dependencies, and post-release defect [17].

What are the known challenges of code refactoring? It is known that certain types of refactoring (API, or structural change) may cause more bugs than regular refactoring [18], [19]. Incomplete and incorrect refactoring can also introduce bugs. According to a report, some automated refactoring tool assistance on IDEs contain bugs [20]. The actual usage of such automated tool facilitates is quite low in developers. One reason is that they are not expressive enough so programmers just ignore the messages [10]. There are also programmers who are not aware of existence of such tools.

What procedure one would follow to form and track the code refactoring decision? The decision is suggested by a few architects over certain potential architectural issues. However, the decisions for inducing, managing, and tracking tech debt are rarely conducted formally. Instead, they are often made by ad hoc. Certain formalization for this process is missing so the impact of technical debt related decision is rarely quantitatively estimated. The history and mental map of product technical debt handling surprisingly rely on 'tribal memory' of team members [8].

3. Methodology

The confusion, hesitation, and reluctance of code refactoring and paying technical debt largely originated from the lack of confidence in technical debt related estimation. Mentioned by programmers and system architects, a legitimate estimation for the impact of a possible code refactoring should be quantitative and also reflect the 'wisdom' from the past decisions. To respond to these requirements, a solid and simple practice is evolved and proposed in this section. The design for the proposed practice is guided by the following several requirements. 1. The practice should include as many factors that dictate the decision making process as possible. 2. The practice should include the weight for the dictating factors leading to final estimation. 3. The practice should be customized according to team characteristics. 4. The practice should be kept as a living document for a development team to accommodate to the climate changes in the development process, which refers to the changes in requirements, time, technical environment, and other unexpected situation. 5. The practice should be archived promptly in order to track the historic information of past technical debt related decisions.

The factors that dictate the decision making process are extracted initially from the real world study cases and categorized. Then, according to the past experiences, the logical connections between these dictating factors are pointed out to establish a branching structure of the decision making diagram. Lastly, a series of rules for adding weight for each decision fork are listed for a development team to follow. The last subsection provides a way to examine the effectiveness of this proposed method according to developer's perspectives, productivity, correctness, defect detection rate, and ease of keeping such a practice as a team routine.

4. Results

4.1. Extracting And Classifying Dictating Factors

From the literature, there are many commonalities in the reported case study regarding the conditions upon which the technical debt decisions were made. In order to build the diagram, those conditions were extracted, simplified, and categorized as the decidingfactors in Table I. Overall, these factors fall into four groups according to their impact on the project. The groups are code base, tool, people (architect/ technical stakeholder, developer, non-technical stakeholder), and outside conditions.

4.2. Finding Logic Connections among Dictating Factors

In fact, many of the dictating factors connect with each other and have a combined impact on the final decision on whether it is wise to perform a code refactoring and pay the technical debt. The organization of the diagram is starting from the left, weighting the decision making process using the results from each group of dictating factors, then exit on the right side with a finalized decision. The diagram shows a skeleton of the practice, and to make the diagram really useful for a development team, the team has to customize it with their value for each factor.

4.3. Customizing Decision Making Diagram

In Fig. 1, each line represents a chance for team voting. After they have voted for the current group, they should have either 'Yes' or 'No' for each group of impact factors. After finishing voting for all categorized group, the decision will naturally be made on the right end by adding the vote on the left side. Also, before entering any voting process, a team should review the diagram and decide the weight for each line according to their characteristics.A survey is composed for this report aiming to evaluate the effectiveness and practical meaning of the proposed diagram and practices. The purpose of this survey is to examine how many of the impact factors mentioned in this study are actually recalled and considered by the surveyed

group in real world software development activities. Also, the surveyed subjects are to provide a score for each of the impact factors reflecting its influence on making the final decision.

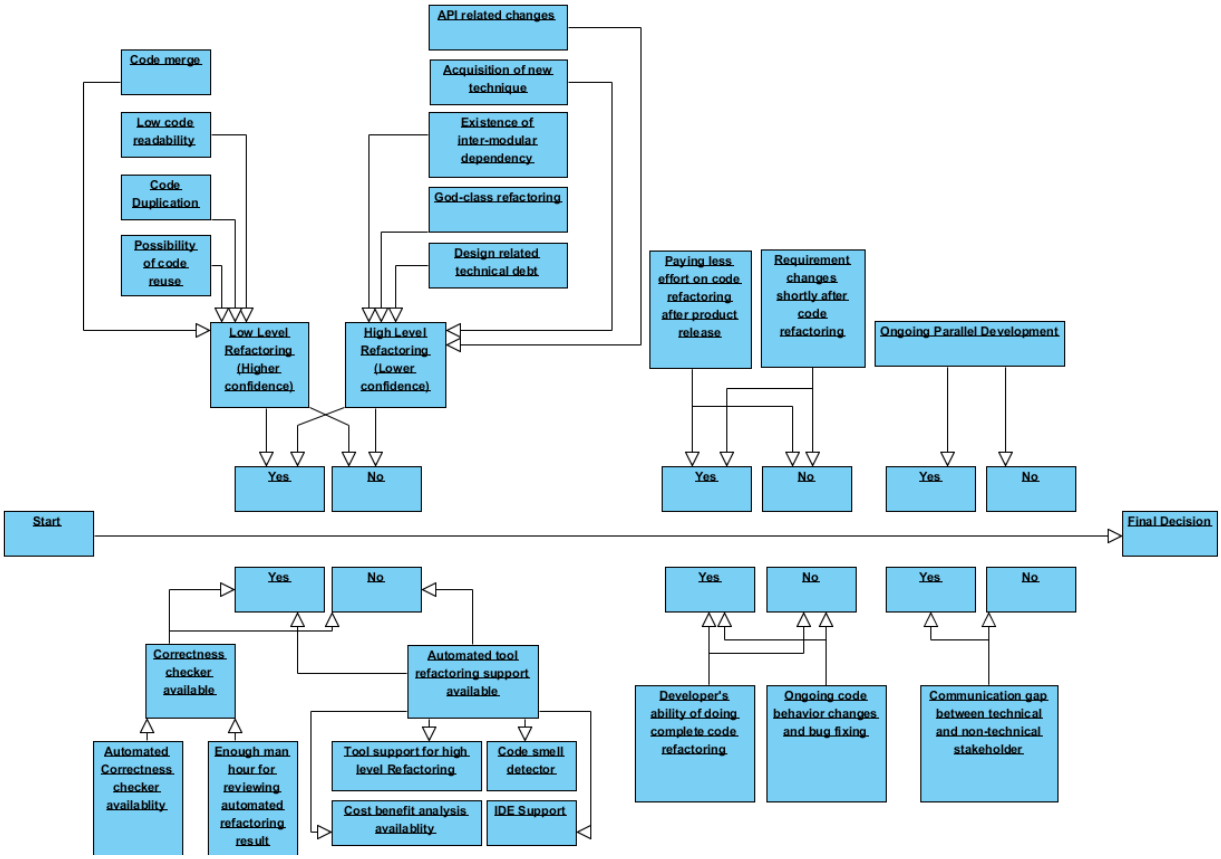

Fig. 1. Decision making diagram integrated with technical debt related dictating factors.

Table 1. Categorized Dictating Factors

CATEGORY	DICTATING FACTOR	SUGGESTION / POSSIBLE OUTCOME
Code Base	Code hierarchies	Pay attention to god-class refactoring Pay design technical debt first
	Code merge	Good chance to refactor, reduce duplication
	Acquisition new technique	Good chance to refactor
	Development environment change	Good chance to refactor
	Low level refactoring	Good chance to refactor
	Low code readability	Sign for refactoring
	Code duplication	Sign for refactoring
	Existence of inter-modular dependency	Sign for refactoring (could be architectural level)
	Reuse code module in different products	Refactor to increase the current code business value
Automated Tools	Cost-benefit analysis	Run an analysis if possible at any time
	Quantitative assessment	Using the provided metrics to estimate the impact
	Correctness checker	Architects are not willing to order code refactoring until knowing there is a way to ensure correctness
	Ide support	Ensure the level of support from before committing using ide refactoring support. It's better not to set expectation too high.
	Specific structural level tool support other than ide	If investing for a specific code refactoring product, make sure it support structural level refactoring.

	Specific code smell tool	Cover more types of code smell, also provide code smell and anti-pattern categorizing to reduce the complexity of detection.
	Man hour for reviewing	If there is no enough resource guaranteeing the refactored code, do not easily use automated refactoring tool.
	Ide/refactoring tool contain bugs	Check tool reliability before integrating it into process
Outside Conditions	Paying less effort on code refactoring after product release	If Such Possibility Is Obviously High, Code Refactoring Is Not 1st Priority
	Requirement changes shortly after code refactoring	If The Possibility Of Requirement Changing Shortly After Code Refactoring Is High, Code Refactoring Is Not 1st Priority
People-Developer	Completeness of refactoring	Incomplete code refactoring could bring more trouble
	Refactoring while making behavior changes/bug fixing?	No
	Awareness of refactoring facilitating tool	Should encourage developers get know the available tool
People-Architect	Communication gap with non-technical stakeholder	Show the potential cost of keeping technical debt inside of the code and clear the communication gap before making decision
	Parallel development	Structurally refactor the project to lower the dependency between team modules
People-Non-Technical Stakeholder	Communication gap with technical stakeholder	Understand the potential cost of keeping technical debt inside the code before making decision

5. Conclusion

The flowchart aims to simplify and standardize the decision making of current technical debt management. Even though the research outcome is not fully confirmed from the survey due to the resource constraint, this research provides an innovativedirectionto work towards another practice fitting into the current Agile methodology.

References

[1] Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Silva, F. Q. B. D., Santos, A. L. M., & Siebra, C. (2011). Tracking technical debt — An exploratory case study. *Proceedings of the 2013 IEEE International Conference on Software Maintenance* (pp. 528–531).

[2] Brown, N., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., & Nord, R. (2010). Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (pp. 47-52). New York: ACM.

[3] Zazworka, N., Shaw, M. A., Shull, F., & Seaman, C. (2011). Investigating the impact of design debt on software quality. *Proceedings of the 2Nd Workshop on Managing Technical Debt (*pp. 17–23*)*.

[4] Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., & Vetrò, A. (2012). Using technical debt data in decision making: potential decision approaches. *Proceedings of the Third International Workshop on Managing Technical Debt* (pp. 45–48).

[5] Cunningham, W. (1993) The WyCash portfolio management system. *ACM SIGPLAN OOPS Messenger*, *4(2)*, 29–30.

[6] Fowler, M. (1999) *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[7] Kim, M., Zimmermann, T., and Nagappan, N. (2012). A field study of refactoring challenges and benefits. *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (pp. 50:1–50:11). New York, NY, USA.

[8] Klinger, T., Tarr, P., Wagstrom, P., & Williams, C. (2011). An enterprise perspective on technical debt. *Proceedings of the 2nd Workshop on Managing Technical Debt* (pp. 35–38).

[9] Ramasubbu N., & Kemerer, C. Managing technical debt in enterprise software packages. *IEEE Transactions on Software Engineering, 40(8),* 758-772.

[10] Murphy-Hill, E., Parnin, C., & Black, A. P. (2012). How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, *38(1)*, 5–18.

[11] Vakilian, M., Chen, N., Negara, S., Rajkumar, B. A., Bailey, B. P., & Johnson, R. E. (2012). Use, disuse, and misuse of automated refactorings. *Proceedings of 2012 34th International Conference on Software Engineering (ICSE)* (pp. 233–243).

[12] Fontana, F. A., Ferme, V., & Spinelli, S. (2012). Investigating the impact of code smells debt on quality code evaluation. *Proceedings of the Third International Workshop on Managing Technical Debt* (pp. 15–22).

[13] Fontana, F. A., Ferme, V., Marino, A., Walter, B., & Martenka, P. (2013). Investigating the impact of code smells on system's quality: An empirical study on systems of different application domains. *Proceedings of 2013 IEEE International Conference on Software Maintenance* (pp. 260–269).

[14] Ratzinger, J., Sigmund, T., & Gall, H. C. (2008). On the relation of refactorings and software defect prediction. *Proceedings of the 2008 international working conference on Mining software repositories* (pp. 35-38).

[15] Beck, K. (2000). *Extreme Programming Explained: Embrace Change.* Addison-Wesley Professional.

[16] Belady L. A., & Lehman, M. M., (1976). A model of large program development. *IBM System Journal*, *15(3)*, 225–252.

[17] Kolb, R., Muthig, D., Patzke, T., & Yamauchi, K. (2006). Refactoring a legacy component for reuse in a software product line: A case study. *Journal of Software Maintenance and Evolution: Research and Practice*, *18(2)*, 109–132.

[18] Kim, M., Cai, D., and Kim, S. (2011). An empirical investigation into the Role of API-level refactorings during software evolution. *Proceedings of the 33rd International Conference on Software Engineering* (pp. 151–160).

[19] Weißgerber P., & Diehl, S. (2006). Are refactorings less error-prone than other changes?. *Proceedings of the 2006 International Workshop on Mining Software Repositories* (pp. 112-118).

[20] Daniel, B., Dig, D., Garcia, K., & Marinov, D. (2007). Automated testing of refactoring engines. *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 185-194).

[21] Murphy, G. C., Kersten, M., & Findlater, L. (2006). How are Java software developers using the Elipse IDE? *IEEE Software*, *23(4)*, 76–83.

Congyingzi Zhang is currently working as an IT advisory consultant in Ernst &Young. She received her M.S. in computer science in 2015 from Bowling Green State University. She has strong research and industry background in project management and software development methodologies/practices. She has publications in multiple computer science research areas and also has presented her work towards professional audience.

Yan Wu is currently working as an assistant professor at Computer Science Department of Bowling Green State University. She was a guest researcher in SAMATE team at NIST. She received her Ph.D. degree in information technology in 2011 from the University of Nebraska. Her research focus is empirical study on analyzing software engineering knowledge in order to support the development and maintenance of reliable software-intensive systems.