Toward Measuring Defect Debt and Developing a Recommender system for their prioritization*

Shirin Akbarinasaji
Data Science Lab
Ryerson University
Toronto, Canada
shirin.akbarinasaji@ryerson.ca

ABSTRACT

Software development managers make a release decision without fully resolving the defects from current and previous releases due to tight deadlines. Deferring the defects would accumulate a tremendous amount of technical debt in the system. Typically, the defect debts are defined as the type of defect that should be fixed. However, due to competing priorities and the limited amount of time and resources, they would be postponed to the next release. In order to aid practitioners ,who make release decisions, to observe the amount of debt, there is a need for quantifying the defect debt. Software bug repositories roughly provide us with information about the amount of time the defect debt exist in the system, the time the defects are resolved and the severity of the defect. We suggest categorizing the defect into the regular defect and debt prone defect by analyzing this information. Afterwards, we compare the regular defect and debt-prone defect to determine the principal, interest and interest probability of defect debt. We also propose the reinforcement learning for scheduling which defect debt needs to be paid and when they need to be paid.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Technical Debt Measurement, Defect Debt, Reinforcement Learning, Software Maintainability

1. INTRODUCTION

Every project requires to be completed and delivered under certain constraints. Project management body of knowledge modeled these constraints as the iron triangle of time, cost and quality (scope). Figure 1 illustrates the project management triangle. Typically, changing one side of the triangle has an effect on the other side of the triangle. For instance, increasing the quality (scope) leads to the growth in time and cost. Similarly, tight time (budget) constraints may cause increasing of the budget (time) and reducing the quality. Complex software projects certainly need to tackle the triple project constraints as well. The quality of software often diminished over time since the software maintenance projects are performed under tight time and resource constraints. The project managers need to make balance between time and cost properties and the required quality level. To achieve the system equilibrium, they might delay some maintenance activities such as documentation, testing or even fixing bugs. The consequence of delaying these technical development activities can be interpreted as a type of debt and it will affect the long term maintenance and development activities. The trade-off between short term benefits of delaying these activities and long-term effect of postponing them is articulated as "technical debt".

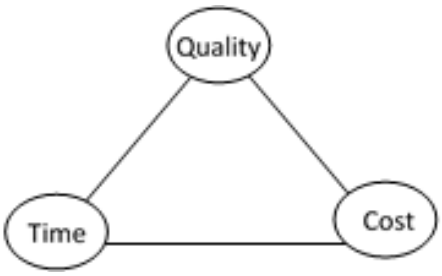

Figure 1: Project Triangle

Generally, the technical debt describes the delayed development activity due to the time and resource constraints. Likewise the financial debt, principal is an amount of effort in terms of time or cost require paying off the debt (i.e complete the task). Interests are the potential penalty in terms of extra amount of effort required to pay as a compensation for what is borrowed [24]. Accumulation of technical debt in the system has tremendous effect on the quality of the system [24]. There is always a choice between paying down the whole principal or continuing to pay off the interest. And, in small software projects, the managers may implicitly decide on the amount of the debt to be paid off and payment

schedule. However, in the large projects, there is a need for a comprehensive system that is able to track and manage the technical debt [19].

Additionally, technical debts may occur intentionally or unintentionally [14]. Intentional debts are the kind of debts that the developers and managers are aware of their existence and they occur deliberately due to the strategic and tactical mission. Unintentional debts occur due to the developers'lack of attention or understanding and the team is not aware of their existence and their location. Furthermore, the debts may also be classified according to its type. According to Li et al. technical debts are classified into 10 types of debts including: requirement, architectural, design, code, test, build, documentation, infrastructure, versioning and defect debt[13]. In this study we particularly focus on defect debts which refer to the defects, bugs or failures found but not fixed in the current release[19].

Since, quantifying the technical debt is a key factor in making decision about incurring, paying off and deferring technical debt, we will initially propose a new approach for measuring the principals and interests of defect debt. Most of the quantifying approaches in the literature are based on the distance between the violation of the code from the ideal code. However, our approach differentiates from the existing approaches since it concentrates on mining of bug repositories and collecting data from issue tracking system.

On the other hand, software development team always faces the high volume of defect reports and change requests in every cycle of release. The challenge is to figure out which instance of defects should be addressed in this release and in which order they are required to be fixed. Therefore, an important next step is recommending a system to the developers and managers to prioritize instances of defect debt in the current release. Our proposed solution for scheduling of the defect debt is based on reinforcement learning. Our ultimate goal is finding an optimal action-selection policy for paying off the defect debt in limited time in such a way that minimizing the amount of interest. We can summarize our contribution as follows:

- Proposing a new approach for quantifying the defect debt,
- Developing an automated framework for prioritization of defect debt

The remainder of the paper is as follows: in section 2, we will briefly explain the current research issues and what we would like to get advice on. Section 3 will review the related work. In section 4, we will specifically discuss our research objectives. Section 5 is a brief explanation of research approach. Finally, section 6 will review next step and section 7 is conclusion and summary.

2. CURRENT RESEARCH ISSUES

Although the technical debt attracts interests of many researchers in recent years, most of the work only capture the theoretical aspect of the technical debt and there is still a lack of empirically based studies in the literate. In this research, we would like to propose an empirical approach for quantifying the defect debt and we seek feedback on our research approach and our proposed solution.

3. RELATED WORK

The technical debt was first introduced two decades ago by Ward Cunningham [3]. He described that "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite". Although it is a fairly recent metaphor in software engineering, it is highly related to well-researched issue like software decay by Lehman and Belady [11] and software aging by Parnas [17] . Software decay is a complexity of software due to continuous changes and software aging is a disability of software to meet required changes. The introduction of technical debt facilitates managing these concepts.

Martin Fowler [7] suggested the quadrilateral categorization of technical debt considering intention (deliberate or inadvertent) versus awareness (reckless or prudent). Brown et al. [1] extended the concept of technical debt from the code level metaphor to architectural and detailed design. Since then, many practitioners and researchers relied on the definition of the technical debt in order to explain various cost-drive issues in software engineering. Tom et al. [24] explored the technical debt concept focusing on dimension of technical debt, the advantages and drawbacks of allowing them in a system and its origins. In another study, Tom et al. [23] did a systematic literature review focusing on the state of academic research for technical debt. Li et al. [13] did a systematic mapping study on technical debt and its management. Seaman and Guo [19] reviewed some issues associated with technical debt and proposed the framework for measuring and monitoring the debt based on risk management approaches. Their framework for managing technical debt was identifying debt, measuring debt and monitoring debt. For identifying debt, especially code-based debt, there are various tools. They are designed to detect code smell [18], modularity violation [25], grime buildup [9] and potential defect. Zazworka et al. [27] found that different tools did not reveal overlapping results, rather they each pointed out a different problem.

Letouzney [12] proposed the SQALE (Software Quality Assessment based on Lifecycle of Expectations) for evaluating the technical debt. Sonar tool[1] is a popular tool for evaluating the technical debt but it is not perfect as it is widely discussed in technical debt literature [5]. Sonar applied quality heuristic for identifying code duplication, violation of code standard, lack of testing, and potential latent bugs. Nord et al. [15] presented a metrics based on architecture and measurement approach for managing the technical debt. Nugroho et al. [16] redefined the technical debt as fixing cost of the technical issue. They performed static analysis to identify debt and estimated the debt principle based on the percentage of changed line code, code duplication, dependency and parameter count and also complexity metric. They calculated the interest estimation by assigning these metrics to risk categories. Guo et al. [8] investigated the effect of technical debt in a real software project by estimating the principal debt according to effort estimation. Zazworka [28] applied cost-benefit analysis for prioritizing

[1]http://www.sonarsource.org/)

the God class debt. He ranked the cost of paying debt and the impact of debt on quality to determine which refactoring activities need to be performed initially. Singh et al. [20] monitored developer activities to estimate the debt interest. Several studies proposed considering interest probability for interest estimation [2], [6]. This is the probability of debt, if not paid, how it would affect other tasks in a negative way [19]. There are also several studies that are presented and classified the technical debt by mapping study by [13].

Xuan et al. proposed the concept of debt-prone bugs and identified three types of debt :tag bugs, reopen bugs and duplicate bugs. Tag bugs are the bugs which "fixme, todo, xxx" tags annotated to them. Reopened bugs are the bugs which are solved by developers but reopen later, and duplicate bugs are the bugs with the same root as existing ones.They conducted a case study on Mozilla to investigate effect of debt prone bugs on software quality. [26].Snipes et al. identified and categorized cost related to fixing the defects or deferring the defects. They stated that decision factor for managing defects from the technical debt perspective are: severity, an existence of a workaround, an urgency of fix required by customer, the effort to implement the fix, the risk of the proposed fix and the scope of testing required[21].

In this study,we particularly concentrate on defect debts and we propose how to measure the principal and interest and interest probability of defect debts. However, the main concern of development team is how to prioritize defect debts in order to minimize the total amount of interest. We propose reinforcement learning to schedule debt prone bugs pay off. For future step, we plan to collect data from bug repository of a real project to show the feasibility of our proposed model.

4. RESEARCH OBJECTIVE

The general objective of this study is to propose an approach that provides the developers and practitioners to make a decision under uncertainty during project management regarding the defect debt. In particular, the goals of this proposal can be summarized as following:

- To propose a simple and straightforward approach for measuring the principal and interest of any defect prone bug instance: In most of the previous studies, the technical debt has been measured by comparing the current state of the code and the ideal target for the code. In this study, we would like to propose a new approach to measure the technical debt by mining the bug repository.
- To develop a framework for automating the prioritization of defect debt: Once technical debt is measured, the next step is to provide a guide to practitioners a decision technique and approach to determine whether or not to pay off the debt at a particular point of time.

5. RESEARCH APPROACH

5.1 Definition of Defect Debt

Defect debts refer to the defects, bugs or failures found but not fixed in the current release, because there are higher priority bugs to fix or there are limited resources to fix them. Jifeng et al. referred to the defect debt by another terminology as "debt prone bugs" and described it as any software bugs which remain in the system because of any immature, incomplete process of fixing the bugs[26].

From the perspective of defect debt definition, the bugs are divided into the bugs which constitute the technical debt and the bugs which do not form any debt. Hereafter, we refer to the latter as regular bugs. Therefore, two type of bugs exist in any bug repositories:

- Regular bugs which refer to the bugs that are submitted and resolved in the same release
- Debt prone bugs which refer to the bugs that are submitted in one release but are not resolved in that release

5.2 Measuring defect debt principal and interest

In management level of any software organization, they are interested in achieving a quantitative understanding of time constraints for fixing the bugs. However, the time constraints that debt prone bugs may impose to the system is totally different from the regular bug fixing time. The fixing time for debt prone bugs is the summation of the standard amount of time required for fixing the bugs and an extra amount of time required to fix them as a penalty of deferring the process. In other word, this fixing process for debt prone bugs may exceed the standard fixing time for regular bugs because the accumulation of unresolved bugs in the system would make it more complicated to repair the defects. Will Snipes et al. identified the following type of time (cost) for fixing defects[21]:

- Investigation time or the time of diagnosing, verifying and finding the alternative solution for defects.
- Modification time which refers to the time of applying the solution to fix defects
- Work around time that deals with providing the bypass for defects which are not resolved immediately
- Customer support time is the time of providing support for the customer because of the defect that exists in the current release
- Patch time or the time of finding temporary solution for fixing the bugs
- Validation time that refers to the time of testing the systems.

Investigation, modification and validation time are incurred for fixing the regular bugs. However, postponing the fixing process of the bugs may impose additional time such as workaround, customer support time and patch time to the system as well. Besides, the investigation cost may increase because of the accumulation of more defects make the diagnosis process more complicated[21]. As mentioned earlier, technical debt principal is the amount of effort in terms of

time that are required for fixing bugs. And the interest is an extra amount of time as a type of penalties which are required for fixing the defect. Consequently, principal value for debt prone bugs may include three components: investigation, modification and validation time. The interest value may compromise additional workaround, customer support time and patch time.

In order to help software development team to better estimate if the defect debt can be absorbed in current release or not, a fair approximation for principal and interest are required. Suppose that two similar bugs with the same characteristic such as severity, same assignment, and same product exist in bug repositories. One bug is treated as a regular bug and one is treated as debt prone bug. For any regular bug, we can easily estimate the regular fixing time by retrieving the historical data from bug tracking system. In any bug tracking system, the bugs are submitted by developers and testers with an ID, description of bugs, its version, the reported date and its severity. The developers who are interested in working on those bugs may assign different labels to bugs such as new, unconfirmed, reopened and etc. They also contain an adequate information regarding the last time the bugs are modified and the history of modification. The fixing time for regular bugs is equivalent to the difference between reporting time of the bugs and the resolving time of the bugs. Therefore, we are able to build a prediction model based on the data from regular bug fixing time to predict the principal for debt prone bugs. The potential input features for feeding the prediction model would be severity, submitter, owner, priority, the indicator if the bug internally discovered or externally discovered, etc.

In order to evaluate the interest for debt prone bugs, initially the variance between the fixing time and the estimated principal needs to be calculated. The interest amount is proportional to this variance. The weighting factor for adjusting that is called severity. Severity is the potential effect of defect in the functionality of the system and customer requirement[21]. For instance, one reason for deferring the bugs fixation to the next release is that the bug is very trivial. Suppose the customer is requesting changes related to background color of the system. In this case, the impact of the defect is a minor irritation. Therefore, the severity of the bug is very low and postponing the defect for many years may not affect the functionality of the system. Therefore, the interest amount for this particular bug is the variance between the regular fixing time and estimated principal times the severity of the bugs. From this point of view, we can conclude that the interest amount is the time difference between principal and real fixing time multiplying the coefficient based on severity. Fundamentally, the more severe bugs pose more interests to the system if they are not fixed in the same release as they reported. Therefore, an interest amount can be calculated as below:

$$InterestAmount = (RealFixingTime - EstimatedPrincipal) * Severity$$

Another component of the interest is the interest probability which refers to the probability that defect debt, if not fixed, will negatively affect the fixing time of other defects[19].For the sake of estimating interest probability, all the bugs that are dependent to current bugs need to be divided by the number of all existing bugs in the current release.

5.3 Applying Reinforcement Learning for Prioritization of Defect Debt

Reinforcement learning is a machine learning task of finding an optimal action-selection behavior of an agent in order to maximize the total amount of reward[22]. Reinforcement learning is inspired by a concept in psychology "reinforcement". In behavioral psychology, reinforcement is a consequence that will motivate the agent behavior by offering specific stimulus followed by the behavior. Basically, reinforcement learning is a problem of an agent who interacts with the environment and tries to achieve an optimal goal. Reinforcement learning has many applications in different domains: robotic control, scheduling, chess playing, backgammon, etc. [10]. Reinforcement learning is based on Markov Decision Process (MDP) and the components of basic reinforcement learning model are as following:[22]:

- a set of actions $a \in A$
- a set of states in the environment $s \in S$
- a set of transactions between the states
- The rules that determine an immediate reward for any transition (r)
- The rules that determine what the agent are able to observe from an environment

The agent interacts with an environment over potentially infinitive discrete time steps of $t = 1, 2, 3, .$ The time step is not necessarily fixed interval time, it can be determined based on point of time that the state change or new action has been performed[4]. At each time, the agent based on state of an environment chooses an action a_t. Then, the agent receives a reward based on chosen action and will be transferred to new state s_{t+1}. The agent can evaluate its performance based on the received reward. The reward defines the goal in the learning problem. The Policy is the rule agent uses to select actions and for each state it assigns a probability to each possible action. The agent tries to adjust policy in order to maximize the accumulated reward [4]. Figure (2) depicts a standard framework of reinforcement learning.

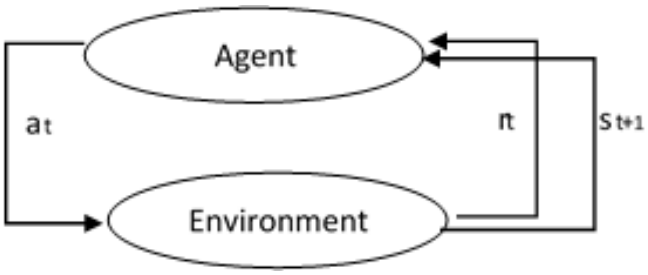

Figure 2: Reinforcement learning Framework

Prioritization of defect debt can be modeled as a reinforcement learning. The main reason is that in scheduling of de-

fect debt, we are facing a changing environment and, therefore, a traditional fixed policy schedule would not work in such a fluctuating environment. Additionally, we are not able to anticipate the long-term consequence of debt prone bugs in the system. In software defect detection, practitioners are looking for recommendations beyond classification. Identifying and recommending a course of action for defect debt is a complex task that traditional classification algorithms may not suffice. A classification algorithm takes a given data to learn and build a model for making a single prediction or decision. However, human learning takes place in three phases, classification, memorization and procedural by interacting with the environment in making decisions. When building recommender systems in complex environments such as software defect debt prediction, we need to use learning techniques such as reinforcement learning where an algorithm considers the effect of actions by trying things to learn. It is different than dynamic programming as we do not know the effects of actions in the case of prioritization of defect debts. Since technical debt is hidden in the code and we do not know its future effect, we need a learning algorithm that considers this uncertainty and learns from its environment by interacting with it. Such learning is more realistic and closer to how humans learn, but it is difficult to implement as a learning algorithm.

The potential framework for prioritization of debt is as follows: Suppose that the developer is the agent in the model that should adjust policy in order to maximize the amount of interest we could save. A set of states in environment might be the amount of time remaining before upcoming release. A set of actions is whether to pay debt i in the upcoming release or not. Reward is equal to amount of interest we save if we pay the debt i at current time instead of postponing it to the next release. Note that after fixing a defect debt bugs, the target is achieving zero (minimum) amount of debt before the release or maximizing amount of interest that we saved. We assume that the developers are fully aware of the current state of the model. They have all the required information about how much debt is in the system and how much time they have to work on debt prone bugs.

Assume that in each release the developer team decides to assign a specific amount of time T to fix some defect debts. At $t = 1$ the developer face totally N defect debts to be fixed. He needs to decide which debt to pay at time $t = 1$. Suppose that the developer decides to fix defect debt i and it would take x_1 minutes to fix it. He would save an interest of $r(s_1)$. At $t = 2$, the remaining time is equal to $T - x_1$. As, defect debt i has been fixed in the system, it would effect the number of defect debt we may face in the system since it will facilitate fixing of some other bugs which are dependent in defect debt i. For instance, the number of defects may change to $N + m - 1$. Suppose, at time $t = 2$ the developer decides to fix defect debt j and it would take x_2. The remaining time would be $T - x_1 - x_2$ and he would save $r(s_2)$. The developer will continue till he reaches time T. He needs to make the decision based on the long-term reward the system would return to him. The expected long-term rewards at each state s for policy ,π, would be:

$$B_\pi(s) = Exp[r(s_t) + \gamma * r(s_{t+1}) + \gamma^2 * r(s_{t+2}) + ...|s_t = s]$$

Where B is the long-term reward in the system, π is the action (policy) the developers decide to take. r is a reward at each state and $0 < \gamma < 1$ is the discount factor which needs to be determined initially. The optimum decision is to optimize the long-term reward:

$$\pi(s) = argmax B_\pi(s)$$

The optimal π would return the sequence of the defect debt that developers should fix to save the maximized amount of interest.

6. PLANNED NEXT STEPS

In order to check the feasibility of our approach, we need to apply it in real world projects and analyze the results. Therefore, we plan our next steps of this research as below:

The initial next step is collecting data from both open source projects and also commercial software companies and applying our proposed techniques for quantifying their defect debt. We also need to consult with developers in each domain to assess our approach. The second step is determining which reinforcement learning method is best situated for our model such as temporal difference learning, Q-learning, SARSA , etc. The last step is to extend our approach to other types of technical debt.

7. CONCLUSION

In this study, we described a new procedure for measuring the principal, interest and interest probability for defect debt. Despite most of the existing approaches which are based on the violation of the program from the right code practice, our proposed method is based on mining the bug repository and standardizing the measurement based on the actual status of the system. Our proposed approach presents a simple heuristic for practitioners with limited knowledge about the architecture of the code in quantifying the defect debt. Typically, we concentrate on the bugs that are opened and resolved in the same release and use them as an original required fixing time. Then, by comparing the defect debt prone to the regular defect, we calculate the interest.

Furthermore, we would like to develop a novel learning framework for scheduling of the defect debt. Our proposed method is reinforcement learning because of the dynamic nature of the defect debt environment. We believe that determining which defect debt to work on is a very complicated problem and developers need an automated framework that suggest them a sequence of defect debt they can fix in limited time that can return the maximum amount of reward to them.

8. ACKNOWLEDGMENT

I would like to express my sincere thanks to Dr. Ayse Bener and Dr. Bora Caglayan for providing me with their valuable advice and guidance. I am very grateful for their support and feedback that help me to improve my proposal.

9. REFERENCES

[1] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, et al. Managing technical debt in software-reliant systems. In *Proceedings of the*

FSE/SDP workshop on Future of software engineering research, pages 47–52. ACM, 2010.

[2] Y. Cai, R. Kazman, C. Silva, L. Xiao, and H.-M. Chen. A decision-support system approach to economics-driven modularity evaluation. *Economics-Driven Software Architecture*, 2013.

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

[4] M. K. P. DARSINI. *Application of Reinforcement learning algorithms to software verification.* PhD thesis, M. Sc. Thesis, Universite Laval, 2006.

[5] R. J. Eisenberg. A threshold based approach to technical debt. *ACM SIGSOFT Software Engineering Notes*, 37(2):1–6, 2012.

[6] C. Fernández-Sánchez, J. Díaz, J. Pérez, and J. Garbajosa. Guiding flexibility investment in agile architecting. In *System Sciences (HICSS), 2014 47th Hawaii International Conference on*, pages 4807–4816. IEEE, 2014.

[7] M. Fowler. Technical debt quadrant. *Bliki [Blog]. Available from: http://www. martinfowler. com/bliki/TechnicalDebtQuadrant. html*, 2009.

[8] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. Da Silva, A. L. Santos, and C. Siebra. Tracking technical debtâĂŤan exploratory case study. In *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, pages 528–531. IEEE, 2011.

[9] C. Izurieta and J. M. Bieman. How software designs decay: A pilot study of pattern evolution. In *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pages 449–451. IEEE, 2007.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, pages 237–285, 1996.

[11] M. M. Lehman and L. A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., 1985.

[12] J.-L. Letouzey. The sqale method for evaluating technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt*, pages 31–36. IEEE Press, 2012.

[13] Z. Li, P. Avgeriou, and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101:193–220, 2015.

[14] S. McConnell. Technical debt. 10x software development. *Blog]. Available at: http://blogs. construx. com/blogs/stevemcc/archive/2007/11/01/technicaldebt-2. aspx*, 2007.

[15] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas. In search of a metric for managing architectural technical debt. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 91–100. IEEE, 2012.

[16] A. Nugroho, J. Visser, and T. Kuipers. An empirical model of technical debt and interest. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 1–8. ACM, 2011.

[17] D. L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[18] J. Schumacher, N. Zazworka, F. Shull, C. Seaman, and M. Shaw. Building empirical support for automated code smell detection. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 8. ACM, 2010.

[19] C. Seaman and Y. Guo. Measuring and monitoring technical debt. *Advances in Computers*, 82:25–46, 2011.

[20] V. Singh, W. Snipes, N. Kraft, et al. A framework for estimating interest on technical debt by monitoring developer activity related to code comprehension. In *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, pages 27–30. IEEE, 2014.

[21] W. Snipes, B. Robinson, Y. Guo, and C. Seaman. Defining the decision factors for managing defects: A technical debt perspective. In *Managing Technical Debt (MTD), 2012 Third International Workshop on*, pages 54–60. IEEE, 2012.

[22] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*, volume 1. MIT press Cambridge, 1998.

[23] E. Tom, A. Aurum, and R. Vidgen. A consolidated understanding of technical debt. In *ECIS*, page 16, 2012.

[24] E. Tom, A. Aurum, and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software*, 86(6):1498–1516, 2013.

[25] S. Wong, Y. Cai, M. Kim, and M. Dalton. Detecting software modularity violations. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 411–420. ACM, 2011.

[26] J. Xuan, Y. Hu, and H. Jiang. Debt-prone bugs: technical debt in software maintenance.

[27] N. Zazworka, C. Izurieta, S. Wong, Y. Cai, C. Seaman, F. Shull, et al. Comparing four approaches for technical debt identification. *Software Quality Journal*, 22(3):403–426, 2014.

[28] N. Zazworka, C. Seaman, and F. Shull. Prioritizing design debt investment opportunities. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, pages 39–42. ACM, 2011.