2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678532

Using Version Control and Issue Tickets to detect Code Debt and Economical Cost

Abdullah Al Maruf
Dept. of Computer Science
Baylor University
Waco, Texas, United States
maruf_maruf1@baylor.edu

Noah Lambaria
Dept. of Computer Science
Baylor University
Waco, Texas, United States
noah_lambaria1@baylor.edu

Amr S. Abdelfattah
Dept. of Computer Science
Baylor University
Waco, Texas, United States
amr_elsayed1@baylor.edu

Tomas Cerny
Dept. of Computer Science
Baylor University
Waco, Texas, United States
tomas_cerny@baylor.edu

***Abstract*—Despite the fact that there are numerous classifications of technical debt based on various criteria, Code Debt or code smells is a category that appears in the majority of current research. One of the primary causes of code debt is the urgency to deliver software quickly, as well as bad coding practices. Among many approaches, static code analysis has received the most attention in studies to detect code-smell/code debt. However, most of them examine the same programming language, although today's software company utilizes many development stacks with various languages and tools. This problem can be resolved by detecting code debt with Issue/Ticket cards. This paper presents a method for detecting code debt leveraging natural language processing on issue tickets. It also proposes a method for calculating the average amount of time that a code debt was present in the software. This method is implemented utilizing git mining.**

***Index Terms*—Code Debt, Code Smell, Architectural Degradation, Technical Debt, Economical-Cost, Version Control, Issue Ticket**

I. Introduction

Many researchers have attempted to classify technical debt using various criteria [1], [2]. For example, Martini *et al.* [3] defined five types of technical debt in their study: code debt, architectural debt, test debt, documentation debt, and infrastructure debt. Regardless of the criteria, code debt is one of the most common categories that was proposed in this study.

Although runtime and static code analysis are viable methods for detecting code debt, they frequently lack support for multiple programming languages. In modern microservice software architecture, it's common to use various languages/stacks to take advantage of distinct benefits. Since 'ticket cards' (e.g., Jira or Trello ticket cards) have become the norm for tracking software development and issues, identifying code debt by processing issue tickets will be compelling.

The cost of a code debt that is paid while the debt is addressed can be estimated using version control software such as git. The purpose of this paper is to demonstrate a way to calculate the cost of code debt and discuss a method for discovering code debt using issue tickets.

This material is based upon work supported by the National Science Foundation under Grant No. 1854049 and a grant from Red Hat Research, https://research.redhat.com.

II. Background and Related Work

Code debt is mostly caused by many types of code smells. Walker *et al.* described their approach of automatically detecting code-smells in a microservice architecture utilizing bytecode and or source code in their study [4].

Faris *et al.* described self-admitted technical debt analysis using code comments with a contextualized vocabulary where the developer writes comments to indicate code debt [5].

In a separate study [6], the authors did a case study on Mozilla and introduced the concept of debt-prone bugs, which they divided into three types: tag bugs, reopened bugs, and duplicate bugs.

Martini *et al.* 's case study [7] presented a measurement technique for architectural technical debt, as well as a mathematical relationship for determining interest in terms of additional work and development.

III. Processing Issue Tickets/Bug Tracking

Natural language processing (NLP) can be used on issue tickets to analyze and detect code debt issues. NLP is an area of Artificial intelligence(AI) that helps machines read language and has applications in spam detection, translation, social media sentiment analysis, and etc. A recent publication of google researchers, BERT (Bidirectional Encoder Representations from Transformers), impacted the NLP community, which is a State-of-the-Art Pre-training model for NLP. NLP can understand contextual relationships between words using this pre-trained machine learning model [8].

A reasonable quantity of train data is necessary to apply the NLP model. The accuracy of the train data determines the likelihood of properly detecting code debt issues. As a result, collecting and verifying train data is critical. We can obtain train data by identifying issues with certain labels and then manually filtering the data.

Popular open-source repositories are a good place to look for code debt issues because they provide best practices for labeling. To collect issues, we can use 'allowed' labels, for example 'bug' and 'todo'. However, we will disregard 'non-allowed' labels while gathering issues with allowed labels. For example, a single issue may contain both the labels "todo" and "documentation". These issues will be skipped in these cases because 'documentation' is not included in the Code debt. Since we cannot rely only on labels, we must manually assess these issues once they have been collected to ensure their accuracy.

After collecting the training data, we can train the NLP model. This can be accomplished using the Bert model that has been pre-trained. With the model trained, it can be applied to other repositories/issue tickets that do not contain any labels.

IV. Economical Cost from version control

The economic cost of technical debt can take several forms, including development hours wasted to address the debt, financial costs incurred resulting from affecting users, a poor basis for development plans, etc. We develop a method to calculate the average amount of time that a code debt was present in the software.

A reference to an updated code commit that fixed the issue/code debt is commonly included in an issue/bug tracking system. We may analyze code changes using this reference to apply the cost estimation method.

To calculate this cost, we start with the commit's code diff. In Listing 1 is described the method in an algorithmic manner. There are three possible scenarios: a code block is only deleted, a code block is deleted due to the addition of a code block, or a code block is only added.

For the deletion code block (scenario 1), we take the sum of all the deleted lines' time differences from the current commit. Then, for each additional line, if it resulted in the deletion of any line (scenario 2), we calculate the time difference between the deleted line and the current commit time. For each additional line, we add this time difference to the entire summation. In the third case, if there is no line deletion but simply addition, we take the average of the upper and lower lines' time stamps of changed blocks and then add the time difference to the overall summation. We take the average after processing all impacted lines by dividing the total summation by the number of affected lines.

Listing 1
Finding Technical Debt cost for each commit

```
sum := 0
Find diff of a commit
For each deleted line:
    sum += time diff of older line commit and
        current commit
For each added line:
    If it caused deletion of line:
        - Use time diff of that deletion time
    Else if there was no deletion:
        - Take the average of upper and lower line
            commit time
    sum += time diff of an old commit and current
        commit
Average cost := sum/#_of_affected_lines
```

V. Conclusions

Code debt detection through an issue/bug tracking system is a good alternative to code analysis for multi-stack microservice architecture since it can be used without depending on the system's programming language. Since NLP and machine learning are becoming more proficient at evaluating data to generate models, this technique may yield superior analysis outcomes. Although gathering data is a challenging effort, it is a do-once task, which means we may reuse the trained model once the data has been collected and the machine learning model is trained. Obtaining a reasonable cost from git mining is also useful for producing matrices that demonstrate which code debts were present in the system for how long.

References

[1] U. Azadi, F. A. Fontana, and D. Taibi, "Architectural smells detected by tools: A catalogue proposal," in *Proceedings of the Scientific Workshop Proceedings of XP2016*, ser. XP '16 Workshops. IEEE Press, 2019. [Online]. Available: https://doi.org/10.1109/TechDebt.2019.00027

[2] R. Roveda, F. Arcelli Fontana, I. Pigazzini, and M. Zanoni, "Towards an architectural debt index," in *2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, 2018, pp. 408–416.

[3] A. Martini, V. Stray, and N. B. Moe, "Technical-, social- and process debt in large-scale agile: An exploratory case-study," in *Agile Processes in Software Engineering and Extreme Programming – Workshops*, R. Hoda, Ed. Cham: Springer International Publishing, 2019, pp. 112–119.

[4] A. Walker, D. Das, and T. Cerny, "Automated microservice code-smell detection," in *Information Science and Applications*. Springer, 2021, pp. 211–221.

[5] M. A. de Freitas Farias, M. G. de Mendonça Neto, M. Kalinowski, and R. O. Spínola, "Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary," *Information and Software Technology*, vol. 121, p. 106270, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584920300203

[6] J. Xuan, Y. Hu, and H. Jiang, "Debt-prone bugs: technical debt in software maintenance," *arXiv preprint arXiv:1704.04766*, 2017.

[7] A. Martini, E. Sikander, and N. Madlani, "A semi-automated framework for the identification and estimation of architectural technical debt: A comparative case-study on the modularization of a software component," *Information and Software Technology*, vol. 93, pp. 264–279, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491630355X

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.