Architectural Degradation and Technical Debt Dashboards*.

Dario Amoroso d'Aragona

Tampere University,Tampere, Finland
dario.amorosodaragona@tuni.fi

Abstract. ***Background.*** Companies frequently try to improve the quality of their software by resolving technical issues that are perceived to have an impact on software quality. Technical information is any information that may be gathered from the source code or the software development process, for instance: code or documentation guidelines, and the use of specific patterns. If these issues are not fixed they may generate technical debt. ***Goal.*** The goal of the Ph.D., which started on January 2022, is to understand which are the causes of Architectural Technical Debt in a real case study and to develop a Dashboard to notify developers as earlier as possible.
Methods and expected contribution We first plan to investigate the actual production code process and then, and then to work with teams to find the best practices and strategies to easily manage Architectural Technical Debt.

Keywords: Technical Debt · Architectural Debt · Software Quality.

1 Introduction

Kruchten et al [4] in their work defined Technical Debt as *"design of implementation constructs that are expedient in short term but that set up a technical context that can make a future change more costly or impossible"*.

The management of Technical Debt is not a trivial task in every context. The field of software quality has reached a good level of maturity and several tools/methods and techniques exist to support developers to manage Technical Debt. However, understanding which are the best practices to adopt in a specific context is challenging. Malakuti et al [8] conduct a study to understand which are the most challenging quest for a company to find the right way to manage Technical Debt. The authors find that the biggest issue is finding the starting point for technical debt management. Other issues that may injure technical debt management are [8]: a) The company's history of quality improvements and its potential (unfavorable) effects on how different people view the efficacy of adopted methods; b) the lack of company-wide understanding of software architecture and technical debt; c) the adoption of technical debt management

* This work is funded by the Doctoral School of Industry Innovation (DSII), and ABB

techniques that may not always produce the desired results; d) the need for quality improvements outside of software, such as process debt.

Technical debt may exist at the architecture level as well as the coding level. Static code analysis tools are available to detect technical debt at code level. Without running the code, these tools can do static code analysis. These tools' primary function is to identify issues in the source code and show developers various metrics that can be used to fix those issues. One of the most well-known static code analysis tools is SonarQube, for instance. [11] defines Architectural Technical Debt (ATD) in software-intensive systems as a metaphor used to describe the "big" design decisions (e.g., choices regarding structure, frameworks, technologies, languages, etc.) that, while being suitable or even optimal when made, significantly hinder progress in the future. In contrast to other types of technical debt, architectural technical debt has an impact on software architecture rather than the code itself. Architectural Technical debt may be produced as a result of bad decision regarding the design of the software architecture, which has a substantial impact on different system components.

In order to early detect Architectural Technical Debt, and to provide dashboards to make possible stakeholders aware of the amount of accumulated Technical Debt, we are planning to investigate the following Research Questions (RQs):

- **RQ1**: Which are the commons causes of Architectural Technical Debt in a company?
- **RQ2**: Which tools/techniques/best practices the company uses to manage Architectural Technical Debt?
- **RQ3**: Which are the effects of these solutions on the Architectural Technical Debt?
- **RQ4**: How is it possible to improve and reduce the Architectural Technical Debt?
- **RQ5**: What visualization techniques can be adopted to visualize the Architectural Technical Debt?

2 Background and related works

Companies typically attempt to enhance the quality of their software by fixing technical problems that are thought to affect software quality. Any information that may be gleaned from the source code or the software development process is referred to as technical information. Examples include the use of particular patterns, adherence to coding or documentation rules, or architectural issues. Such problems cause technical debt if they are not resolved.

Alves et al. [1] proposed a mapping study to identify the various type of Technical Debt and the known solution adopted for each of these. Their results show how Architectural Technical Debt refers specifically to those problems encountered in the architecture of the system (e.g., high coupling, high cohesion, no modularity). These types of issues may affect architectural requirements (performance, robustness, among others).

Brown et al. [3] first and Krutchen et al [5] after in their works have investigated the relationship between the Code Debt [1] and the Architectural Debt. The results show how this type of debt cannot be resolved with simple refactoring in the code but needs more extensive development activities.

Recently, Malakuti et al. [8] have investigated the Technical Debt in their company finding that the cause of Architectural Technical Debt is to be attributed to different factors: *(i)* Changes in Context: changes in the business context, aging technology, ad-hoc approach for supporting multiple hardware/software variants, natural evolution and aging; *(ii)* Business: time and cost pressure, requirements shortfall, misalignment of business goals; *(iii)* Processes and Practices: insufficient processes, insufficient documentation, inadequate software engineering practices, and tools, inadequate planning; *(iv)* People: inexperienced teams, unclear quality-related roles, and responsibilities, insufficient motivation concerning quality improvement, coordination and communication shortfall, and lack of common understanding of technical debt.

Moreover, Malakuti et al.[7] have investigated which are the open challenge in the management of Technical Debt reporting how there is still a lack of a clear taxonomy of debt and their relations, how it is possible to introduce systematic management of each technical debt category, and finally how to manage the relations between the technical debt issues between the technical debt categories. Lenarduzzi et al. [6] proposed a systematic literature review on Technical Debt Prioritization, the findings demonstrate how software companies are under increasing pressure to provide value to customers. Finding a compromise between allocating developer time, effort, and resources to new feature implementation and TD remediation tasks, bug repair, or other system upgrades is crucial. In the work proposed by Besker et al. [2] this concept is summarized as "the pressure of delivering customer value and meeting delivery deadlines forces the software teams to down-prioritize TD refactoring continuously in favor of implementing new features rapidly" Budget, resources, and available time are crucial considerations in a software project, especially at the prioritization stage, as investing time and energy in refactoring tasks typically means that less time can be allocated to other tasks, such adding new features. This is one of the key causes, along with their frequent focus on delivering customer-visible features, explaining why software companies don't always allocate additional funds and resources to fixing Technical Debt in the source code [12].

Additionally, according to Martini, Bosch, and Chaudron [9], TD refactoring activities typically have lower priority than the development of new features, and Technical Debt that is not directly connected to the deployment of new features is frequently put off.

Vathsavayi and Syst [10] echo this notion, stating that "Deciding whether to spend resources for developing new features or fixing the debt is a challenging task." The researchers highlight that software teams need to prioritize new features, bug fixes, and TD refactoring within the same prioritization process.

As the final results of the works proposed by Lenarduzzi et al. [6] the authors demonstrate how studies frequently concentrate their prioritization techniques

on prioritization among various TD components, with the aim of determining which item should be refactored first. None of the prioritizing strategies mentioned in the publications analyzed directly discusses how to prioritize between putting time and effort into reforming TD and developing new features.

3 The proposed approach

The Ph.D. plan consists of the four following steps:

1. Analysis of Architectural Technical Debt in literature
2. Analysis of the strategies/best tools/practices adopted in the company (ABB) to manage Architectural Technical Debt
   - 2.1 Analysis of ABB code production process and comparison with the state of the art
   - 2.2 Analysis of the tools actually used in the CI process.
   - 2.3 Survey with the developers to investigate why they used that specific tool or why they don't use any tool.
   - 2.4 Work with teams to understand how to mitigate the issues that arose.
3. Investigation on the effects of the solutions adopted
4. Implementation of a tool/dashboard that can help the developers to manage technical debt showing the metrics more useful for them according to the previous results.

Step one attempts to answer RQ1. RQ2 is answered in the second step. The third step try to answer RQ3 and RQ4. Finally, the last step answers RQ5.

3.1 Research Methodology

Step 1 The first step is to analyze the Technical Debt in literature, understanding which is the starting point reached now, which are the most common causes of Architectural Technical Debt identified in these last years, which are the best practices identified, and how they are applied in different contexts. The goal is to study in deep all the aspects of Technical Debt, considering all the aspects of the coin. Thus, the issues related to the code, to the architecture, to the lack of knowledge, to the lack of best practices but also the aspect related to the social smells and how this impacts the introduction of Technical Debt in a system.

Step 2 The second step is related to the state of the art of Technical Debt management in the company (ABB). Thus, understand which are the best practices and the tools yet used to take under control the Technical Debt in their code, focusing on which of these tools can be used also to manage Architectural Technical Debt. This step is composed of 4 phases:

- *Phase 1* In the first phase we will perform a manual analysis of their repositories, of the Continuous Integration tools used, and of the configuration of the pipelines to understand which is their production process;

- *Phase 2* In the second phase we will perform a manual analysis of the Continuous Integration tools (e.g., Jenkins, Azure pipelines) to understand which tools they have in their production process to manage Technical Debt (e.g., Lattix, SonarQube, Test framework, Lint);
- *Phase 3* The third phase consists to dispense a survey to the developers to understand why some tools are used at the expense of others, why some teams do not use any tools (if there are), if there is some correlation between the product that the teams are developing and the tools used by them, if there is some adopted tool useful to manage Architectural Technical Debt (eg., to prioritize ATD items; to minimize ATD in source code) and so on;
- *Phase 4* Finally we want to select a subset of developer teams to work with them improving their Technical Debt management.

Step 3 In the third step, we will evaluate the solutions adopted to understand if they really improve the management of Technical Debt or not. In the latter case, we plan to investigate other solutions, introduce them, and evaluate them. In other words, steps 2 and 3 will be performed in an iterative cycle.

Step 4 Finally, in the last step, we intend to collect all the metrics of the tools introduced in the previous steps that are really useful for developers and show them in a dashboard.

4 Expected Contributions

Our research wants to help developers to manage (eg., prioritize/minimize) Architectural Technical Debt. In the long-term, we expect to find which are the best practices that a company with the same characteristic as the company investigated by us can introduce to take under control the Architectural Technical Debt in their products. Moreover, we guess that the findings of our research will help the community to better understand how Technical Debt grows up in a real case study, which is the impact on a real production process, which solutions can be adopted, which could be the causes of Technical Debt in a company.

References

1. Alves, N.S., Mendes, T.S., de Mendonça, M.G., Spínola, R.O., Shull, F., Seaman, C.: Identification and management of technical debt: A systematic mapping study. Information and Software Technology **70**, 100–121 (2016), https://www.sciencedirect.com/science/article/pii/S0950584915001743
2. Besker, T., Martini, A., Bosch, J.: Technical debt triage in backlog management. 2019 IEEE/ACM International Conference on Technical Debt (TechDebt) pp. 13–22 (2019)
3. Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., Zazworka, N.: Managing technical debt in software-reliant systems (2010)

4. Kruchten, P., Nord, R., Ozkaya, I.: Managing technical debt: Reducing friction in software development (2019)
5. Kruchten, P., Nord, R.L., Ozkaya, I.: Technical debt: From metaphor to theory and practice. IEEE Software **29**(6), 18–21 (2012)
6. Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., Arcelli Fontana, F.: A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. Journal of Systems and Software **171**, 110827 (2021). https://doi.org/https://doi.org/10.1016/j.jss.2020.110827, https://www.sciencedirect.com/science/article/pii/S016412122030220X
7. Malakuti, S., Heuschkel, J.: The need for holistic technical debt management across the value stream: Lessons learnt and open challenges (04 2021)
8. Malakuti, S., Ostroumov, S.: The quest for introducing technical debt management in a large-scale industrial company. In: Jansen, A., Malavolta, I., Muccini, H., Ozkaya, I., Zimmermann, O. (eds.) Software Architecture. pp. 296–311 (2020)
9. Martini, A., Bosch, J., Chaudron, M.: Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. Information and Software Technology **67**, 237–253 (2015). https://doi.org/https://doi.org/10.1016/j.infsof.2015.07.005, https://www.sciencedirect.com/science/article/pii/S0950584915001287
10. Vathsavayi, S., Systä, K.: Technical debt management with genetic algorithms. 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA) pp. 50–53 (2016)
11. Verdecchia, R., Kruchten, P., Lago, P., Malavolta, I.: Building and evaluating a theory of architectural technical debt in software-intensive systems. Journal of Systems and Software **176**, 110925 (2021), https://www.sciencedirect.com/science/article/pii/S0164121221000224
12. Vidal, S., Vazquez, H., Diaz-Pace, A., Marcos, C., Garcia, A., Oizumi, W.: Jspirit: a flexible tool for the analysis of code smells. pp. 1–6 (11 2015). https://doi.org/10.1109/SCCC.2015.7416572