The Quest for Introducing Technical Debt Management in a Large-Scale Industrial Company

Somayeh Malakuti[1(✉)] and Sergey Ostroumov[2]

[1] ABB Corporate Research Center, Ladenburg, Germany
somayeh.malakuti@de.abb.com
[2] Softability Group Oy, Helsinki, Finland
Sergey.Ostroumov@abo.fi

Abstract. The long lifetime and the evolving nature of industrial products make them subject to technical debt management at different levels such as architecture and code. Although the classical steps to perform technical debt management are known, in a study that we have been performing in a large-scale industrial company as our client, we realized that finding a starting point, which leads to the desired outcome, is in fact a major challenge. This paper elaborates on various causes that we have identified for this challenge, and discusses our stepwise approach to address them so that the software quality can be improved. We believe that our experiences can be beneficial for both practitioners and researchers to gain more insight into applying quality improvement in practice as well as indicating open areas for further research.

Keywords: Quality improvement · Technical debt management · Software architecture · Source code analysis

1 Introduction

Technical debt is defined 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" [1]. Technical debt management can start by identifying technical debt, followed by measuring its impacts, prioritizing, documenting, repaying/preventing, and in parallel monitoring it [2]. Various proposals exist to implement each of these activities in practice [3,4].

The long lifetime and the evolving nature of industrial products (e.g., robots, controllers, sensors), as well as the usual time to market requirements of industrial companies, make the industrial products subject to technical debt at various levels. Since the field of software architecture has reached a level of maturity such that several commercial tools, methods and techniques exist to support practitioners, one might be tempted to start by adopting these in the companies. However, based on our experience, we realized that it is in fact a major

A. Jansen et al. (Eds.): ECSA 2020, LNCS 12292, pp. 296–311, 2020.
https://doi.org/10.1007/978-3-030-58923-3_20

challenge to find a starting point for technical debt management, which leads to the desired outcome.

We identified that there are various issues that contribute to this challenge. Examples are: a) The history of quality improvements in the company and its possible (negative) impacts on the perception of different people of the effectiveness of adopted methods, b) Lack of common understanding of software architecture and technical debt across the company, c) Adopting certain technical debt management approaches that would not necessarily lead to the desired outcomes, d) Necessary quality improvements beyond technical debt in software, for example, process debt, infrastructure, and hardware debt, and e) Unforeseen situations such as the impacts of COVID-19 on the availability of resources and on their priorities.

Using an illustrative case study that we have performed as consultancy for one of our clients, we explain that a stepwise method must be taken to address these issues and to introduce technical debt management in a company. Our case study is a project that has started in 2018, and so far consisted of three phases.

The first phase was focused on high-level identification of technical debt via assessing the modularity status of a pilot case. The second phase was about repaying technical debt via architecture and code refactoring. However, in the second phase we could not achieve our refactoring goal, because the adopted technical debt management approach was mainly based on gut feeling and domain experience rather than objective and systematic means. Nevertheless, this experience was important to raise awareness of systematic technical debt management approaches, as well as gaining more insights into possible causes of technical debt in the company. To perform systematic technical debt management in the third phase, we first identified various kinds of debt (e.g., technical, process, infrastructure) and their possible impacts on software quality improvement activities. Although it is not possible to address all debt at once, we explain our strategy to narrow down the scope to some feasible steps to be able to still proceed with technical debt management, while coping with the unforeseen changes in the company as well as the impacts of other debt on our activities.

Based on our previous experiences and this case study, we observe that most large-scale hardware-oriented industrial companies have more or less similar characteristics in terms of their time to market pressure, distribution of teams, software architecture competence, etc. Therefore, we believe that this paper can be beneficial for both practitioners and researchers to gain more insight into challenges in applying technical debt management in practice, as well as identifying open areas for further research.

This paper is organized as follows. Sect. 2 provides an overview of the project and its phases; Sect. 3 explains our experience in the phase 1 for identifying modularity issues; Sect. 4 summarizes our experiences in adopting a not very systematic technical debt management approach; Sect. 5 outlines our vision for a more comprehensive technical debt management methodology; Sect. 6 summarizes various challenges and lessons learned; Sect. 7 provides related work and our insight on open research areas; Sect. 8 outlines conclusion and future work.

2 Project Setting and Methodology

The illustrative case study is a project in the form of external consultancy that we have been offering to an industrial company as our client since 2018. The company naturally has stronger background in hardware design, but software has gained strong importance over the time as well.

Throughout the project, eight software developers/architects, one process manager and one technology manager closely participated in the case study. In addition, the project results were frequently presented to larger audience in the company. The software engineering competence differed among the participants.

Since the project team was geographically distributed, we have adopted different approaches such as frequent teleconferences, face to face workshops, field observation, interviews and surveys to reach the goals of the project.

Before the start of the project in 2018, there were several internal workshops in the company with the aim of achieving bug-free software. The result of those workshops was that insufficient modularity of the software is the main reason for the increasing number of bugs as well as long maintenance time. As a result, the project started in 2018 with the aim of validating this hypothesis and providing support for improving the modularity of the software in three phases.

Since we needed to gain deeper insight into the software and the way it was developed, the methodology that we adopted in the project was not fixed at the beginning; throughout the project we defined our next steps based on the learnings of the previous steps. Our activities can be summarized as follows:

Assessing the Modularity Status of the Software: The phase 1 of the project was about getting to know the software better and validating the hypothesis of our client regarding the low modularity of their software. As a result, we identified various kinds of technical debt that exist at the architectural and code levels.

Prioritizing and Repaying Technical Debt at the Architecture and Code Levels: Refactoring large-scale legacy software as a whole require significant amount of time and resources, which could not be offered by our client. Therefore, our client opted for an iterative approach of refactoring at architecture and code levels. Here, the key challenge was to identify architecturally-significant requirements for each iteration, and to improve their design and implementation. In the phase 2, a high-level sketch of new architecture was defined by our client, and we were requested to refactor two pilot modules based on the sketched high-level architecture.

However, this approach was not as effective as we expected, for example, because the actual time spent for refactoring was twice longer than the estimated time and still the code did not fully adhere to the envisioned architecture. Inadequate technical debt quantification and prioritization, as well as several other issues were among the reasons that we could not reach our refactoring goals.

Identifying the Root Causes of Software Quality Issues: Based on our field observation, we realized that there were multiple attempts in past to improve the

quality of the software; however, the quality has eventually dropped over time. This observation besides our experience in the phase 2 motivated us to first identify various kinds of debt that exist (e.g., technical, process, infrastructure) and their root causes. Even if we could not resolve all the root causes at once, this study would help us be more aware of their impacts on technical debt management, and prioritize technical debt more effectively. Therefore, we gained more insight into these, in the phase 3 via field observation, interviews, workshops and a survey.

Adopting a More Systematic Technical Debt Identification and Prioritization Approach: As the result phase 2, we realized that the distance between the current architecture and the envisioned one is too large, and it is necessary to define intermediate architectures to reach to the envisioned one in an iterative manner. Each intermediate architecture must address certain technical debt, and must accept some other technical debt that will be addressed by future intermediate architectures. The phase 3 of the project focuses on defining such intermediate architectures, and adopting the state-of-the-art methods for identifying and prioritizing technical debt, while taking other kinds of debt and their impacts also into account.

The details of these phases and their results are explained in the subsequent sections.

3 Phase 1: Modularity Assessment

Since the company considered insufficient modularity as the main technical debt, we were requested to perform an initial study on modularity status of a pilot part to validate this hypothesis.

As the first step in this phase, via a requirement elicitation workshop with 15 participants, we collected the stakeholders' requirements regarding the modularity of the software. The requirements were assessed and classified from various perspectives, such as developers/architects, customers, and product owners. Examples of the collected requirements are: a) Reducing the required time for new developers to learn the software, b) Reducing the number of new bugs introduced during the maintenance phase, c) Flexibility in configuring the software for each customer, and d) Flexibility in deploying software modules on different processors.

Then a specific subsystem was nominated by the architects as the pilot case. The lines of code in the pilot case change over the time, as it is being actively developed and refactored. In 2018, it contained around 300K lines of code in C/C++. To understand the current architecture of the pilot case, we made use of the available documentation, which explained the modules and their responsibilities. The documents also described the data flow among the modules. Alongside the documents, we adopted various static code analysis tools such as Understand [5], TeamScale [6] and CppDepend [7] to recover the architecture of the software.

Various metrics such as afferent (incoming) coupling, efferent (outgoing) coupling, module size, and circular dependency were assessed using the aforementioned tools. Figure 1 shows a high-level call graph of the pilot case, which is generated using the Understand tool. Here, the anonymized boxes depict various subsystems, and the links among them show the dependencies.

Our analysis revealed several architectural and code smells, such as high number of circular dependencies, high number of fan-in and fan-out, high degree of code duplicate, excessive usage of global variables, God classes, etc. Moreover, it was clarified that no specific architecture pattern or principle was followed in the pilot software.

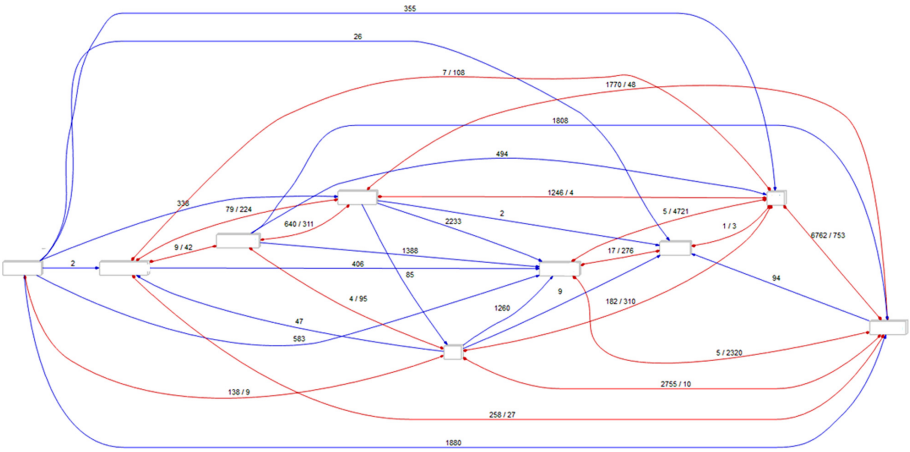

Fig. 1. The dependency graph of the pilot part

4 Phase 2: Modularity Improvement

As a result of the phase 1, the architecture team in the company provided the high-level sketch of a new layered architecture to be used for further refactoring activities. Here, the assumption was that due to the complexity of the software, the large degree of modularity issues, as well as shortage of resources, it was not feasible to have a fully detailed architecture at once. In this phase, we were asked to support by the refactoring of two pilot subsystems, mainly at detail design and code levels.

We defined four goals for this phase: a) Setting up the refactoring infrastructure consisting of suitable tools and metrics, and means for lean architecture knowledge management, b) Adopting and validating these for refactoring two pilot subsystems, c) Defining domain-specific design patterns for improving modularity at the code level, and d) Knowledge transfer via frequent teleconferences and the two-week visit of the consultancy team to the company for face to face architecture discussion/refactoring and another requirements solicitation workshop.

4.1 Tool Selection

There are many static code analysis tools available, which usually support different metrics and detect different issues in the code. We relied on existing studies on assessing these tools [3,8] as well as our field observation in the company to define criteria for selecting a suitable tool for practical usages. Some important criteria are listed below:

- Support for architecture recovery and analysis using different metrics: The company already had one static code analysis tool installed in its build system, which was not actively used by all teams due to its false positive results, among other barriers. Therefore, we decided not to introduce yet another tool that performs analysis at the code level. Nevertheless, since there was no up-to-date architecture picture of the software available, the adoption of a tool that supports architecture recovery and analysis seemed to be necessary.
- Support for stepwise refactoring: In large-scale software system, refactoring will take place in small steps, during different timeframes, by different groups of architects who are expert in specific parts of the software. Therefore, the tool must enable each architect to iteratively apply refactoring only at the desired parts of the software. Naturally, local refactoring is not always sufficient, and the tool must also enable architects to refactor the overall architecture of the software.
- Support for delayed code-level refactoring: The tool must enable the architects to refactor the architecture and the structure of the code, without the need for immediately changing the actual code. This is an important criterion because of two reasons: a) due to resource and time constraints, software developers who should implement the refactoring may not immediately be available, b) since different parts of the architecture are designed by the architects, who may work at different parts of the company with different refactoring priorities, the tool must enable collaborative refactoring at the architecture level before changing the code.
- Support for the integration in the Jenkins' build system: The tool must be integrated in the build system of the company so that architectural metrics and rules can frequently be validated and monitored.
- No reliance on the Git history: Some tools rely on the Git history of commits to identify modularity violations. Such features assume consistent tagging of each Git commit by developers in terms of bug fixes, feature development, refactoring, etc. However, such consistent tagging may not exist across all projects in the company, especially in older projects.
- Consistency of adopted tools in the company: To make sure the project results are transferable to other divisions of the company, we aimed at keeping the adopted tools across divisions consistent.

We selected Lattix [9] among several other tools [5–7,9–12] that we assessed. Tools such as Understand [5], CppDepend [5], SonarQube [5] and TeamScale [6] are mostly at code level, and do not fully fulfill our above-listed criteria. DV8 [11] is an architecture assessment tool; however, it heavily relies on Git history of commits, also the soundness of its metrics is still under validation [13].

4.2 Architecture Knowledge Management and Documentation

The requirement specifications, architectural models and decisions were documented in different formats in the company, although not very consistently nor comprehensively. For example, requirements on each task were documented as part of the task description in Azure DevOps; Microsoft Visio was adopted for architectural modelling, but the models were outdated; Atlassian Confluence was used for collection of technical debt; decision forces are not documented at all, or very inadequately.

We assessed the lean ways of documenting architecture knowledge as well as architecture models. Our study mainly targeted available proposals for Architecture Decision Records [14], as well as other templates such as arch42 [15]. We also assessed various modelling tools such as Enterprise Architect, IBM Rhapsody and Structurizr [16]. Our assessment of the tools was eventually postponed to future, because the requirements of the company on such tools were not clear. Nevertheless, we tailored a template for architecture decision records and adopted it for documenting our new design.

4.3 Refactoring Two Subsystems

The refactoring of two subsystems took almost six months, where we focused on the detail design of two subsystems, proposed new module structures and interfaces, implemented the new modules, and adjusted test cases and build scripts. In this activity, we explored various design patterns that were suitable for the two subsystems, and assessed the alternatives based on various quality attributes such as readability, modularity and testability. Due to confidentiality reasons, we cannot share the details of this activity.

In summary, the major achievements of the phase 2 were: a) The installation of Lattix in the build system of the company, b) The identification of various design patterns for refactoring two subsystems, c) The selection of most suitable design patterns based on various quality attributes, d) Refactoring of the pilot modules to some degree, and e) Gaining more insight on the obstacles to perform technical debt management more effectively.

5 Phase 3: Systematic Technical Debt Management

The decision of our client to put more emphasize on code-level improvements can be interpreted as an example of technical debt prioritization. However, this prioritization was not as effective as we expected, because: a) The actual time spent for refactoring was twice longer than the estimated time, b) Some modularity-related technical debt was resolved in one subsystem, and more or less similar debt had to be introduced in another subsystem, c) The refactored code was not fully integrated into the final software release, d) The proposed tools and documentation templates were not actively used later.

Nevertheless, phase 2 helped us illustrate that a) effective technical debt management at code level cannot be achieved if technical debt at architecture

level is not adequately addressed, b) although gut feeling and domain experience are also important in technical debt prioritization, we need to complement them with more systematic approaches, c) there are several other kinds of debt in the company, which impact software quality improvement activities.

Based on our field observation during the phase 2, as well as some internal workshops and interviews with various participants, we developed a set of hypotheses about various categories of debt, their root causes, and their impacts on software quality improvement activities. We started the phase 3 of the project by performing a survey in the company to validate our hypotheses based on the responses of a larger set of colleagues. The survey aimed at collecting information on the impacts of technical debt on the daily work of developers, an estimation of technical debt growth rate, the causes of technical debt, other kinds of debt (e.g., process and infrastructure) in the company, as well as the self-assessment of developers of their software engineering competences.

There were 100 respondents in total, consisting of 70 software developers or architects, and 30 participants in different managerial roles. The survey was active for two months; afterwards we analyzed and normalized the results and presented to large audience in the company. The survey confirmed that most of the participants see technical debt as an obstacle in their everyday work, and technical debt is increasing in various parts of the software. Also, the majority of the participants assessed themselves to have intermediate knowledge of various software engineering topics.

5.1 Causes of Technical Debt

Due to confidentiality reasons, the survey details cannot be published. Nevertheless, based on the results of the survey, we classified the causes of the technical debt as below. This classification is an extension of the proposal by [1].

Changes in Context: This category contains following causes:

- Changes in business context: New hardware products are introduced, but old software is reused and adapted for them. This results in the replication of technical debt across multiple products.
- Aging technology: Where there is good potential for adopting new technologies such as software product lines, old technology is still used for programming, which limits the usage of advanced object-oriented concepts as well as reuse across different products.

  In addition, there is technical debt in the build system, due to the adopted ad-hoc approach for supporting multiple hardware/software variants.
- Natural evolution and aging: Some parts of the software have been developed and extended for many years, with a lot of patches to accommodate new use cases.

Business: This category contains the following causes:

- Time and cost pressure: There is time pressure for developing new features. In addition, cost limitations for hardware development forces developers to resolve some hardware-related issues at the software level; i.e. technical debt in hardware propagates to software.
- Requirements shortfall: Customer requirements are not well-documented and accessible to the developers. The missing history of the requirements makes it rather impossible to identify obsolete parts of the code that are no longer requested.
- Misalignment of business goals: The focus of each team is usually on developing and optimizing a single feature in the code that may be shared across different hardware products. There is no systematic way of aligning and ensuring quality across different hardware products.

Processes and Practices: This category contains the following causes:

- Insufficient processes: Although it was confirmed by the participants that over the past years the organizational/software processes have improved, there are still various possibilities for further improvements. For example, current software processes do not include technical debt management and topics related to it such as templates for technical debt backlog items, quality checklists for assessing technical debt, etc. In addition, current processes to enable systematic cross-team architecture and code review can significantly be improved.
- Insufficient documentations: Missing or outdated architecture and code documentations make it very hard and or even impossible to keep track of previous design decisions. Besides, product roadmaps are not always in hand to help architects identify future changes of the architecture and prepare the architecture to accommodate those changes. As a result, changes are applied in a rather ad-hoc way, leading to more technical debt.
- Inadequate software engineering practices and tools: Current practices for requirements management, architecture design, coding and testing should be updated based on latest developments in respective areas; for example, via adoption of new tools, requirements engineering methods, ATAM (architecture trade-off analysis method), etc.
- Inadequate planning: To achieve desired goals in technical debt repayment, the extent of required refactoring in architecture, code, test cases and build scripts should be identified and considered during the planning phase. Otherwise, as we experienced in the phase 2 of the project, not all planned technical debt can be repaid, or even new technical debt may be introduced due to the time pressure.

People: This category contains the following causes:

- Inexperienced teams: Software competences may not be very strong in companies that historically produce hardware products. Although there are already

multiple trainings going on in various areas, it was mentioned that such theoretical trainings must be combined with more practical coaching to become more effective.
- Unclear quality-related roles and responsibilities: There is no clear role in teams for performing quality checks and improvements. Although some colleagues have the role of software architect, they may spend the majority of their time on bug fixing.
- Insufficient motivation with respect to quality improvement: Some participants feel demotivated regarding this topic because of the large number of identified causes as well as previous attempts for improving the quality, which had not fully achieved their goals.
- Coordination and communication shortfall: There are different Scrum teams that focus on different parts of the software within or across multiple products. To manage intra-, and inter-products dependencies, systematic means of coordination and communication across teams must be in place. Otherwise, developers must attend all the meetings of other teams to get informed on relevant decisions, and this is impractical. Likewise, to manage inter-products dependencies and to derive course-grained and futuristic architectural decisions, systematic means of coordination and communication among teams and portfolio managers must be in place. Currently, such communications are not performed systematically, causing teams make sub-optimal decisions based on the information that is available to them.
- Lack of common understanding of technical debt: Some developers believe that technical debt only exists in the older parts of the software, and the growth rate of technical debt in the new parts of the software is fixed or even reducing. However, the above-listed causes of technical debt do not make new software parts immune against technical debt.

5.2 Various Aspects of Technical Debt Management

Based on our survey, there was consensus that debt is not limited to code and architectural levels, and other kinds of debt such as social, process and infrastructure debt also impact the overall quality of products. Accordingly, we derived Fig. 2 that depicts various elements, which we believe play a role in achieving successful debt management at the system level.

The impacts of debt are visible in evolving products, which need to accommodate new customer requirements in certain period. Therefore, clear understanding of current and possibly futuristic customer requirements helps to plan for the evolution of the products better. Likewise, business visions define the roadmaps for future changes of the products, and availability of such roadmaps help architects to prepare the architecture for accommodating the future changes.

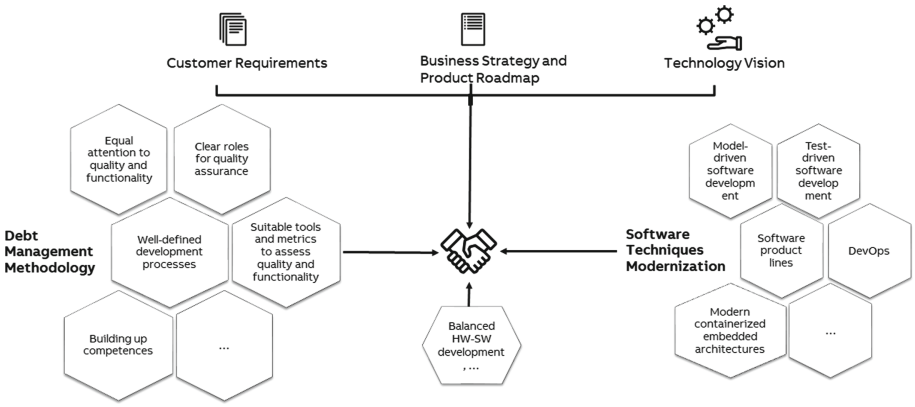

Fig. 2. The scope of technical debt management

Software engineering topics and software technologies are frequently evolving to address the need of industries. Consequently, companies must now and then modernize their technologies and methods to benefit from these advances. Methods such as test-driven software development, software product lines and model-driven software development pave the way to prevent various technical debt.

A systematic debt management methodology must become a mandatory part of a company. Depending on the causes of debt in the company, such a methodology must cover various aspects such as defining suitable processes, competence development, and infrastructure support.

Last but not least, industrial devices are 'systems' that contain both hardware and software components. Therefore, debt management can only be effective if its scope is at the system level, so that the interplay between technical debt in mechanical, electrical software architecture can be taken into account [17].

5.3 Towards Systematic Technical Debt Identification and Prioritization

Naturally, the depicted scope of debt management in Fig. 2 is very large, requires significant investment of time and money, and more importantly requires agreement and approval from different parts of the company.

Since it usually takes some time to achieve such an agreement, we had to narrow down the scope in the phase 3 of the project to the parts that are feasible in short-term. We see this scope definition as a kind of debt prioritization, where we have to identify our focus on the debt that can be paid considering the available constraints. Nevertheless, the awareness about various kinds of debt and their causes helps us mitigate their impacts on the selected scope of work.

We decided to proceed with more systematic architectural-level technical debt identification and prioritization based on available proposals in the literature [3,4,18–20], and adjusting them based on the needs of the company.

We are pursuing the following goals in this phase: a) Defining intermediate architectures to reach to the envisioned one in an iterative manner, b) Proposing a catalog of criteria for more objective prioritization of technical debt based on existing proposals in the literature, c) Quantitative and qualitative assessment of intermediate architectures and the required refactoring effort via Lattix and the ATAM method, d) Defining criteria of the 'Definition of Done' for the respective refactoring activities, e) Proposing a unified template for documenting technical debt at the architectural and code levels, f) Extending existing infrastructure (e.g., Azure DevOps) to incorporate the template, and g) Defining necessary extensions to the existing processes to incorporate the above steps. The results of this experiment will be reported in our future papers.

6 Challenges and Lessons Learnt

Below is the summary of our observed challenges and lessons learnt so far:

Assessing the Product Development Maturity Level: We believe that assessing the maturity of the development processes within a company should be one of the early activities towards software quality improvement in the company. The Capability Maturity Model [21] could be taken as a reference for this matter. Such an assessment helps to gain deeper insight on the impacts of current processes on (technical) debt accumulation.

Finding the Right Balance to Repay Various Debt: It is a well-known believe that software architecture may reflect the organizational structure of a company; meaning that there are more than just technical factors that influence the design of software architecture. Our study also confirms that to achieve the desired outcome in software quality improvement, one has to identify both technical and non-technical debt and their causes (e.g., see Sect. 5.1), identify their impacts on each other and develop a holistic approach to address debt.

Since we cannot address all causes of debt at the same time, there will always be some debt remaining at different levels, which we may revisit in the future if their impacts are high enough. For example, in the phase 2 of our project, it was clear that there is debt at the documentation, architecture, requirements, infrastructure, social and process levels. However, addressing them requires major agreements and effort across the company, which cannot be combined with refactoring the code in parallel. Even if we keep the focus on technical debt management, finding effective ways of prioritizing technical debt at architecture and code levels is already very challenging. Therefore, one must always assess such cases and select the most effective path to follow, knowing that some decisions have to revisited later when other debt is addressed or even is accepted to remain.

Improving Quality-Aware Working Culture: Introducing a methodology for systematic technical debt management in a large-scale company requires adjusting

the working culture of the company, which is inherently a long-term activity. Most developers feel the impacts of technical debt on their work, but may not agree on the root causes of it. Especially if there are many causes of technical debt, it is always a challenge to find the right starting point. A major challenge here is to achieve common understanding of technical debt management and its associated topics across the company, as well as raising awareness on the role that each person can play to improve the situation. In addition, allocating dedicated time for quality improvement (e.g., 20% of developers time) is not sufficient, and more support (e.g., competence development, processes and infrastructure improvement) should be provided to help developers use that time more effectively.

Considering Previous/Ongoing Quality Improvement Attempts: In large-scale companies usually there have been several attempts in the past, or even there are ongoing activities for improving the quality of products. This can include introducing a tool that offers some metrics for technical debt measurement, attempts to improve the processes, attempts to harmonize architecture across multiple products, etc. To successfully contribute in the quality improvement activities, one has to familiarize himself with such activities and their positive/negative impacts on the perception of different parties on the various attempts.

Bi-directional Stepwise Competence Development: Introducing yet another tool or method, and providing some theoretical trainings are not enough for developing the competence of teams. Practical coaching to integrate new tools and methods in the working culture of the company is rather mandatory. Based on our experience, a stepwise approach is needed to illustrate the effectiveness of different quality improvement approaches, and to validate the effectiveness of the state-of-the-art methods in practice. This also requires the familiarity of the consultancy team with the daily working culture of the company to be able to effectively adjust existing methods to fit the specifics of the company.

Dealing with Frequently Changing Scope: In working with large-scale companies, we often hear from managers and developers that 'we have to start somewhere'. Finding the right starting point for technical debt management, which can have the desired impacts under dynamically changing external and internal conditions, is a major challenge. For example, where it was already very difficult to receive agreement to invest more time on technical debt management, the impacts of COVID-19 on the availability of resources made it even more difficult. Therefore, we had to adjust the scope so that we would not need extra resources allocated to our project. Considering that introducing systematic technical debt management in a company is a long-term activity, one has to always be prepared for adjusting the scope of the work to cope with such uncertainties.

7 Related Work

Various studies [22,23] have been performed to provide a better understanding on how teams in large-scale agile organizations coordinate, and show that the

coordination of work between teams influences teams' internal processes and how each team makes decisions. The study in [24] focuses on identification of social and process debt in agile companies. The study in [25] confirms that agile practices has positive impact in managing technical debt. We also believe that effective technical debt management at code and architecture levels cannot be achieved if social and process debt is not tackled.

In [4], the authors propose an approach to adopt business process management (BPM) to make the technical debt prioritization decision process more aligned with business expectations. Based on our insight discussed provided in our paper, we believe that the study proposed by [4] should be extended for large-scale embedded systems, where there are different business goals for software and hardware components, as well as there are multiple product variants with not completely aligned business goals.

Systematic technical debt prioritization based on multiple criteria is an active area of research [3,18–20]. In [18], the authors define multiple criteria based on customer, project and nature of technical debt for prioritization of technical debt. In [19], the authors investigated how a model of cost and benefits of incurring technical debt could be part of the change control board's decision process. In [20], by studying multiple large-scale companies, various criteria such as customer needs, lead time, cost/benefit, maintenance costs, and violated quality rules are determined to be considered for technical debt prioritization.

Where the input from these studies are useful for the phase 3 of our project, we found the focus of these studies to be limited to technical debt only, without taking other kinds of debt such as social and process debt into account. To the best of our knowledge, such a comprehensive study of debt prioritization is still missing in the research community. Besides, the cross-disciplinary management of technical debt by considering the impacts of mechanical, electrical and software engineering aspects of technical debt is still open for further research.

8 Conclusions and Future Work

Developing highly reliable software with low maintenance costs is ultimately the goal of most companies. Since software engineering is an evolving field, there is always the likelihood that advanced methods and techniques are not fully adopted in large-scale companies, which have long-living software with strong time to market requirements. Therefore, if a company requires support to improve the quality of its software, one may quickly strive for adopting various advanced software architecture and coding topics in the company. However, there are usually many challenges and obstacles along the way, which one has to find an effective solution to cope with.

In this paper, we reported on our ongoing experience in introducing technical debt management in a large-scale industrial company as our client. We explained that even if the classical steps for technical debt management are known, one has to take a stepwise approach to build common understanding on the relevant topics inside the company, to validate different approaches of technical debt prioritization, and to deal with various challenges that appear along the way. As for

future work, we will continue assessing and adjusting our current methodology based on the lessons that we learnt along the way. In addition, we would like to adopt existing metrics and invest on various company-specific metrics to monitor quality improvement trends at various levels such as planning, architecture, code, and testing.

References

1. Kruchten, P., Nord, R., Ozkaya, I.: Managing Technical Debt: Reducing Friction in Software Development. Addison-Wesley Professional, Boston (2019)
2. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**, 193–220 (2015)
3. Lenarduzzi, V., Besker, T., Taibi, D., Martini, A., Fontana, F.A.: Technical debt prioritization: state of the art. A systematic literature review, ArXiv, vol. abs/1904.12538 (2019)
4. de Almeida, R.R., Kulesza, U., Treude, C., Feitosa, D.C., Lima, A.H.G.: Aligning technical debt prioritization with business objectives: a multiple-case study (2018)
5. SciTools - Understand. https://scitools.com/
6. TeamScale. https://www.cqse.eu/en/products/teamscale/landing/
7. CppDepend. https://www.cppdepend.com/
8. Fontana, F.A., Roveda, R., Zanoni, M.: Technical debt indexes provided by tools: a preliminary discussion. In: 2016 IEEE 8th International Workshop on Managing Technical Debt (MTD), pp. 28–31 (2016)
9. Lattix. https://www.lattix.com/
10. Structure101. https://structure101.com/
11. Cai, Y., Kazman, R.: Dv8: automated architecture analysis tool suites. In: 2019 IEEE/ACM International Conference on Technical Debt (TechDebt) (2019)
12. SonarQube: SonarQube. https://www.sonarqube.org/
13. Nayebi, M., et al.: A longitudinal study of identifying and paying down architecture debt. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 171–180 (2019)
14. Parker-Hernderson, J.: Architecture Decision Record (ADR). https://github.com/joelparkerhenderson/architecture_decision_record
15. arch42. https://arc42.org/
16. structurizr. https://structurizr.com/
17. Dong, Q.H., Ocker, F., Vogel-Heuser, B.: Technical debt as indicator for weaknesses in engineering of automated production systems. Prod. Eng. Res. Devel. **13**, 273–282 (2019). https://doi.org/10.1007/s11740-019-00897-0
18. Ribeiro, L.F., Souza Rios Alves, N., Gomes De Mendonca Neto, M., Spínola, R.O.: A strategy based on multiple decision criteria to support technical debt management. In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 334–341 (2017)
19. Snipes, W., Robinson, B., Guo, Y., Seaman, C.: Defining the decision factors for managing defects: A technical debt perspective. In: 2012 Third International Workshop on Managing Technical Debt (MTD), pp. 54–60 (2012)
20. Besker, T., Martini, A., Bosch, J.: Technical debt triage in backlog management. In: IEEE/ACM International Conference on Technical Debt (TechDebt) (2019)
21. Paulk, M.C., Weber, C.V., Curtis, B., Chrissis, M.B.: The Capability Maturity Model: Guidelines for Improving the Software Process. Addison-Wesley Professional, Boston (1994)

22. Dingsøyr, T., Moe, N.B., Fægri, T.E., Seim, E.A.: Exploring software development at the very large-scale: a revelatory case study and research agenda for agile method adaptation. Empir. Softw. Eng. **23**(1), 490–520 (2017). https://doi.org/10.1007/s10664-017-9524-2
23. Bjørnson, F.O., Wijnmaalen, J., Stettina, C.J., Dingsøyr, T.: Inter-team coordination in large-scale agile development: a case study of three enabling mechanisms. In: Garbajosa, J., Wang, X., Aguiar, A. (eds.) XP 2018. LNBIP, vol. 314, pp. 216–231. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-91602-6_15
24. Martini, A., Stray, V., Moe, N.B.: Technical-, social- and process debt in large-scale agile: an exploratory case-study. In: Hoda, R. (ed.) XP 2019. LNBIP, vol. 364, pp. 112–119. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-30126-2_14
25. Holvitie, J., Leppänen, V., Hyrynsalmi, S.: Technical debt and the effect of agile software development practices on it - an industry practitioner survey. In: 2014 Sixth International Workshop on Managing Technical Debt, pp. 35–42 (2014)