The Strategic Technical Debt Management Model: An Empirical Proposal

Chapter · May 2020

DOI: 10.1007/978-3-030-47240-5_13

CITATIONS
3

READS
654

2 authors:

Paolo Ciancarini
University of Bologna
350 PUBLICATIONS **4,465** CITATIONS
SEE PROFILE

Daniel Russo
Aalborg University - Copenhagen
49 PUBLICATIONS **462** CITATIONS
SEE PROFILE

The Strategic Technical Debt Management Model: An Empirical Proposal

Paolo Ciancarini[1(✉)] and Daniel Russo[2]

[1] University of Bologna - Italy and Innopolis University, Innopolis, Russia
paolo.ciancarini@unibo.it
[2] Department of Computer Science, Aalborg University,
Selma Lagerlöfs Vej 300, 9000 Aalborg, Denmark
daniel.russo@cs.aau.dk

Abstract. Increasing development complexity in software applications raises major concerns about technical debt management, also in Open Source environments. A strategic management perspective provides organizations with an action map to pursue business' targets with limited resources. This article presents the Strategic Technical Debt Management Model (STDMM) to provide practitioners with an actionable roadmap to manage their technical debt properly, considering both social and technical aspects. To do so, we pursued a theoretical mapping, exploiting a set of interviews of 124 carefully selected and well-informed domain experts of the IT financial sector.

Keywords: Technical debt · Strategic management · Empirical software engineering

1 Introduction

Software development is a complex social task, undertaken by groups of people who have to cope with existing legacy requirements, which need to evolve according to market expectations. Software is rarely developed from scratch, and its design integrity is shared among different people; moreover it changes in time, according to new and unpredictable requirements.

A primary concern of each organization which uses software is typically related to the maintenance of its assets, and the evolution of its products to deal with market competition [8]. This is the most common situation were shortcuts are undertaken for several reasons, typically related to budget or schedule constrains. Long-term software maintainability is often neglected, and a short-term perspective is pursued. Architectural layering and code smells are the typical results of such management, which drifts to poor software quality. This kind of pattern is well-known: it was named by Cunningham as *Technical Debt* (TD) [4]. A growing community holds that software quality practices to improve systems'

Published by Springer Nature Switzerland AG 2020
V. Ivanov et al. (Eds.): OSS 2020, IFIP AICT 582, pp. 131–140, 2020.
https://doi.org/10.1007/978-3-030-47240-5_13

sustainability (e.g., refactoring) is ultimately a business decision [10]. Even in the domain of open source software there is a trend into exploiting the concept of technical debt as intentional, hence strategic: see for instance the study on self-admitted technocal debt found in [7].

A support model is needed to allow developers and managers to make choices, i.e., whether or not to refactor [18]. The idea itself of *debt*, taken from finance, implies to manage with limited resources and make the best out of it. This has the following implications.

Firstly, it relates to a rational **trade-off** between quality and budget. This idea is deeply rooted in software managers. Maturity models (e.g., CMMI) are valuable frameworks to predict software quality according to the minimal software process requirements. Typically, for critical software, a CMMI level of 4 or 5 is required by contract. It should assure the customer that the software house undertakes full refactoring. Accordingly, the price per LOC incorporates this effort, increasing the value of software. On the contrary, to secondary software systems, which will be replaced soon, will be devoted less budget. Indeed, there is no economic rationality to assign the same relevance to a system's application or component. Thus, in a situation of limited resources, ones have to make choices and assign priorities. This means that not all software components will receive the same amount of resources, leading to different levels of quality within the same information system. Hence, it needs to be adequately managed, making such choices rational. Technical debt may arise from random processes and poorly managed development. Therefore, a management model helps to identify the sources of technical debt, so to assign them the respective priority.

Secondly, it is **quantifiable**. Once you take a bank loan, you have to repay it with interest rates. Interests are not always negative when the need is to deploy some secondary code good-enough-to-work and focus on strategic applications. Nevertheless, it should also be clear that, at a certain point in time, one has to repay the debt with its interests.

Thirdly, it permits to **make investments**. We do not always have enough budget or time to implement all requirements. However, if stakeholders consider one application of strategic value to exploit competitive advantages, organizations need to make investments. So, they will earn much more than they have to pay for interests. Subsequently, software houses are willing to deploy some applications, also with high technical debt, since the earnings from their use will fully repay their later refactoring, letting make them profit.

Fourth, **debt management** needs to be considered. Issuing any debit note has its cost because it implies some operations which have to be undertaken to grant the loan. Typically, the investment needs to be planned, the debt traced, repaid, and managed. It is the same for software projects. Once the need for investments emerges, what does it mean in terms of technical debt management? So, questions like *where* is the debt, *how* and *when* are we going to repay it, and *who* should do that - have to be taken into great care.

In our research journey, we experienced all those issues in highly complex banking information systems [16,17]. Also, we modeled those concerns in an ontology to make such knowledge representation inter-operable with other similar systems [1].

It is quite usual for a large organization to exploit different software houses for the evolution and maintenance of the information systems. This is a typical situation were technical debt emerges for several reasons:

- Architectural stratification as a consequence of a lack of conceptual integrity.
- Applications become rapidly outdated, afflict by high maintenance cost, and difficult to evolve.
- Market competition pushes for new applications.
- Core System Optimization is costly, and the skills to evolve old mainframes are lacking.

The management of an information system's quality is, with such a setting, a challenging effort. Still, it is a pivotal task.

From a software management perspective, the four outlined financial implications suggest that the literature about technical debt [9,12] is unable to provide enough explanatory power of this phenomenon since the social part is missing. The reason is that technical debt is a socio-technical matter, and as such comprehensive handling is needed to address its complexity. In the end, managing technical debt means to control people.

To get a deep understanding of this crucial issue, we developed an innovative research design and involved 13 top managers of the IT banking industry for the items identification and 124 IT banking domain for the item validation and construct definition. We asked them to outline the most relevant *software quality concerns* related to banking information systems. We obtained 28 unique factors through the Delphi-like research design, which we mapped in the proposed managerial model.

The main contribution of our paper is the Technical Debt Management Model, gathered by highly relevant empirical research to provide the practitioner's community with a valuable tool to manage technical debt in a structured way.

In the rest of this paper, we will contextualize the metaphor of technical debt through the Agile triangle in Sect. 2. Moreover, we will briefly explain how we elicited software quality concerns by a high-level panel of banking industry experts in Sect. 3. After mapping the items, we propose a model to manage technical debt in Sect. 4. Finally, we conclude our study in Sect. 5.

2 The Technical Debt Triangle

The idea of technical debt is not a radically new one. In software engineering, we are well aware of software deterioration problems, where the complexity of software raises along with its evolution [11], and reuse [2,3]. Similarly, software ages when it is not able to cope with new requirements due to irreconcilable technology paradigms [15].

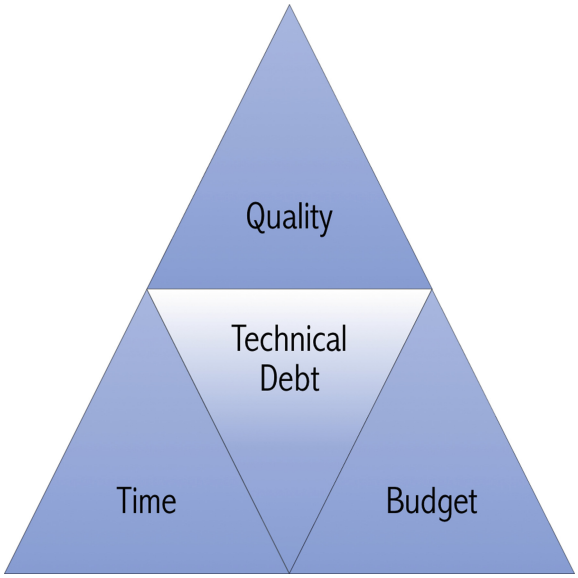

Fig. 1. Technical Debt Triangle

To better explain the concurrent drivers of technical debt, we elaborate on the idea of the Agile Triangle [6]. The three different dimensions of technical debt are budget, time, and scope. They are conflicting since it is not feasible to develop high-quality software with a low budget in a short time. This assumption is the baseline for any strategic software development model since you typically have to deal with limited resources. In an ideal situation, where resources are unlimited, there is no need for any strategic management effort, since strategic choices are simply not necessary. So, if we are focused on one corner, we are going to weaken the other two. If we are concerned about the scope, the relevant budget must be planned and sufficient time should be devoted to the project. If we need some new functionalities in a short time, it will have a high cost (since unplanned tasks require the organization to reschedule the work-flow), and quality will suffer from necessary fine-tuning before deployment, e.g., minor bug fix or refactoring. Finally, in case of a low budget, the project will last reasonably for an extended period, since few people can work on it. Moreover, the quality will also probably decrease since the most skilled (and paid) developers will work on other projects (Fig. 1).

These last two cases, which are the most frequent ones, typically lead to technical debt. Often, technical debt is beneficial because it permits to make investments in case of a project's budget, which is lower than that effectively needed. It is a continuous trade-off between long and short term perspectives. However, it needs to be a strategic (i.e., rational) decision. Often, technical debt is caused by subsequent uncontrolled, unplanned, and irrational tasks, which led to an explosion of its interests. At that point, organizations typically struggle

because they feel to be on a sinking ship, were new applications just increase the debt, and no exit strategy is planned because they often do not know where to start to repay it [16].

A strategic management model is a roadmap, where any organizations, according to its internal and external constraints, plan the (i) what, (ii) how, (iii) when, and (iv) where the technical debt should be managed.

Several mapping studies of the literature have been pursued, to identify and analyze the elements required to manage technical debt [5]. These studies have proposed some taxonomies, which could be useful to understand the most impelling issues while dealing with technical debt. However, a taxonomy is not a management model.

Other scholars introduced a maturity model for technical debt management, where they identified three levels of awareness in software factories; see for instance [20]. Still, a proposal for a strategic management model is missing.

3 Research Design

Defining technical debt is hard [10]. In our research we identified the proxy-construct of *Software Quality Concern*. Our domain experts were able to express openly, in a structured scientific procedure, all concerns regarding the software quality of the information systems they were working on.

To do so, we first identified 13 top managers of the IT financial industry to cover in, a representative way, the entire the addressed problem. Those experts were able to elicit several concerns, and, using the Delphi methodology, they were able to reach full consensus about the solicited items. Afterward, we identified other 124 domain experts through a stratified random sampling technique and asked them to express their level of agreement with the proposed items and to add personal opinions on every single item.

In that way, we were able to identify the 28 concerns, namely: (1) Module interfaces complexity, (2) Interfaces architectural complexity, (3) Custom software quality, (4) Increase of maintenance costs, (5) Quality vs. Time & Budget, (6) Quality vs. System analysis, (7) System analysis vs. Documentation, (8) Documentation vs. Time & Budget, (9) New packages functionalities vs. complexity, (10) Packages vs. Documentation, (11) Packages documentation vs. System analysis, (12) Application & Maintenance contracts vs. Documentation, (13) International applications vs. Quality & Maintainability, (14) Domestic applications vs. Quality & Maintainability, (15) Measurement of software quality, (16) Lower developers' expertise and professionalism, (17) Contracting & Skills, (18) Lacking tools & Methodologies, (19) Establishment of internal and external development processes, (20) Developer's professionalism vs. Skills, (21) Developer's professionalism vs. Rates, (22) Web technologies vs. Methodologies, (23) Quality vs. Requirements, (24) Requirements vs. Methodologies, (25) Requirements vs. Technical jargon, (26) Data analysis vs. Functional analysis, (27) Functional analysis vs. Data modeling, (28) Documentation standards and tools. For a better understanding of the concerns and the research design, refer to [16,17] (Fig. 2).

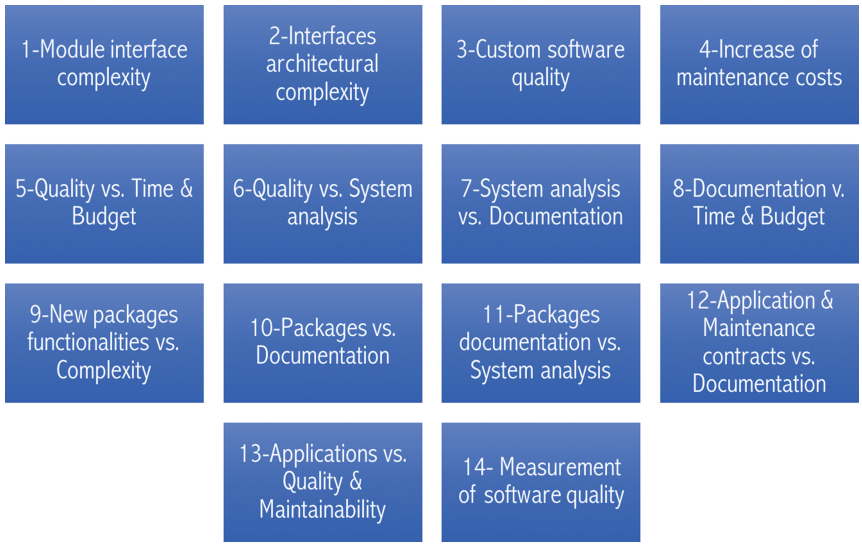

Fig. 2. Software Quality Concerns

After the elicitation and validation of the relevant concerns, we mapped them within an established technical debt management taxonomy [13]. Interestingly, the technical one has already been explored and led to the development of the SQALE method [12]. Indeed, SQALE identifies several technical sub-dimension and their related software metrics. However, we did not find in the scientific literature any similar framework addressing the social dimension. Therefore, we proposed a comprehensive managerial model. We pursued a theoretical mapping of the items within their related sub-dimensions. The outcome of the theoretical mapping for both technical and social dimensions is listed in Table 1.

Table 1. Technical Debt type mapping

Social		Technical [13]	
TD type	Items	TD type	Items
Staffing & Seniority	20, 21	Requirements	6, 14, 23, 24, 25
Skills & Training	16, 17, 21, 22	Design & Architecture	1, 2, 4, 6, 9, 11, 26, 27
Risk & Contracting	3, 5, 12, 17, 19, 20	Code	1, 4, 5, 10, 19, 23
Stakeholder involvement & Outsourcing	14, 17, 19, 25	Test & Defect	3, 4, 5, 6, 13, 15, 16
		Build & Versioning	4, 15, 18
		Documentation	4, 7, 8, 10, 11, 12, 28

Finally, we were able to relate the empirically gathered items within a relevant managerial concept, which is the base of our Strategic Technical Debt Management Model, described in the next section.

4 Strategic Technical Debt Management Model

The essential elements of any strategic management model are: situation analysis, strategy formulation, strategy implementation, and evaluation & control [19]. Such a model is typically non–linear since the surrounding environment changes more or less rapidly according to exogenous market-related factors. Consequently, we build our STDMM on such assumptions, fine–tailoring the evaluation and control phase, as represented in Fig. 3.

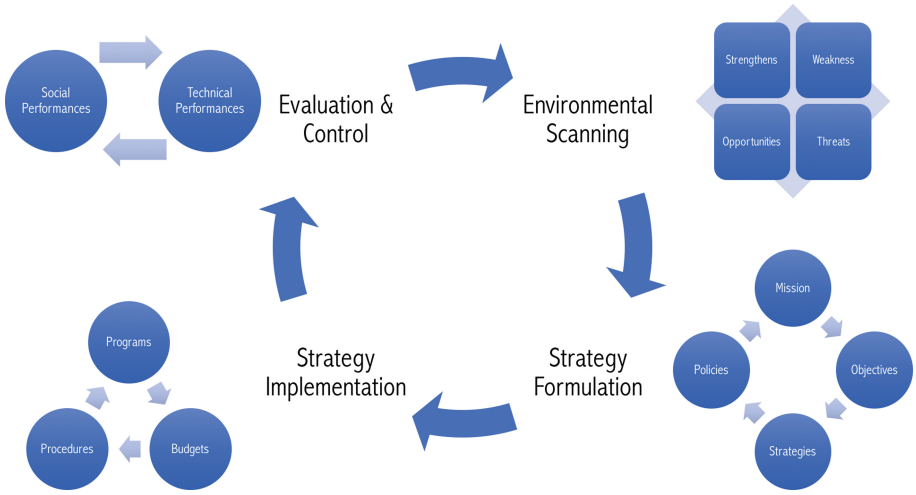

Fig. 3. Strategic Technical Debt Management Model

Thus, to strategically manage technical debt means to take into account:

1. **Situation analysis (Environmental Scanning).** The first step should raise awareness within the organization about its external and internal environment. Benchmarking competitors is always a good idea since there is no such absolute baseline. Indeed, markets that experience continuous requirements adaptations at high velocity may generate more TD concerning those who have to follow standardized quality processes (e.g., CMMI). Therefore, identifying *External Opportunities & Threats* supports effectively management decisions. This enables an organization to position itself to the market. Relevant questions might be: how does the market manage TD? Or where do competitors invest their debt? Afterward, *Internal Strengths & Weaknesses* have to be analyzed. In particular: how good is my organization in managing TD, or where go my investments, are fundamental questions to scan the internal environment. Typically, the first quantification of the already existing debt happens in this phase.

2. **Strategy Formulation.** After the assessment phase, the organization should draw its strategy to manage technical debt. Each strategy has four main pillars [19]. To make it self-evident, we propose key questions, useful to draw a strategy:
   - *Mission*: why should an organization exploit TD?
   - *Objective*: which results do we want to accomplish, and by when?
   - *Strategy*: which plan do we define to achieve mission & objectives?
   - *Policies*: which internal guidelines for TD decision making do we want to have?

   Of course, these are high-level questions, which need to be tailored to any organization. However, they provide good–enough fit for every organization which aim is to manage its TD strategically.
3. **Strategy Implementation.** Once the planning activity has been concluded, the strategy should carry on.
   In particular, specific *programs* and activities needed to manage effectively, TD has also to be outlined and followed. Continuous code inspection with quality metrics is a valuable example of this step.
   These kinds of plans have a price, which needs an ad hoc *budget*. Notably, every activity with no budget is poorly effective; thus, every organization should allocate enough resources. Otherwise, an STDMM is rather useless.
   Finally, *procedures*, intended as a sequence of steps needed to manage TD, are also part of the model. The aim is to leverage on TD to finance urgent software development needs or to repay it. However, *how* to manage these decisions in a complex organization is not trivial. Therefore, internal procedures guide both developers and management to use TD strategically.
4. **Evaluation & Control.** The last phase accounts for the continuous follow up of the strategy. Although the technical dimension of TD is a well known one [14], we introduced here also social aspects, which are equally important for an effective management strategy. We were able to elicit and validate through or research journey [16,17] relevant social dimension regarding TD management, which is described in Table 1.
   In particular, *staffing and seniority* impacts on TD, since a right mix of senior and junior developers enhances code quality. Moreover, the project team should be as stable as possible in time, as also suggested by Brook's law. *Skilled developers* on *ongoing training* are educated to deal with new complex tasks and technologies. For example, the presence of a training plan is a positive software quality indicator. *Risk mitigation through contracts*, which transfer TD to contractors (like Service Level Agreement), is a common practice for most organizations. These aspects poorly relate to technical assessment techniques of TD, although they have an impact on organizations. Thus, to effectively manage TD means to include risk and contracts within the strategy. Finally, any software is a collective product, which can be developed in different ways (e.g., internally or externally to the company). The involvement of *stakeholders*, especially *outsources* is a key quality issue. Continuous interaction is an effective way to have a complete overview of the development process and the use of TD. This is even more important for an outsourced project, where the customer typically has poor visibility about what is happening.

5 Conclusions

To conclude, this paper provides three main contributions. Firstly, it maps the social dimension of TD. Secondly, it proposes a strategic approach to manage TD. Finally, STDMM is a first attempt to include the social dimension of TD within a strategic technical debt management model.

Future works will focus on the extension and validation of the proposed model.

Acknowledgments. This work was partially funded by the Institute of Cognitive Sciences and Technologies (ISTC) of the Italian National Research Council (CNR), and the Consorzio Interuniversitario Nazionale per l'Informatica (CINI).

References

1. Ciancarini, P., Nuzzolese, A.G., Presutti, V., Russo, D.: Squap-ont: an ontology of software quality relational factors from financial systems. arXiv preprint arXiv:1909.01602 (2019)
2. Ciancarini, P., Russo, D., Sillitti, A., Succi, G.: A guided tour of the legal implications of software cloning. In: Proceedings of the 38th International Conference on Software Engineering Companion, ICSE 2016, pp. 563–572. ACM (2016)
3. Ciancarini, P., Russo, D., Sillitti, A., Succi, G.: Reverse engineering: a European IPR perspective. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1498–1503. ACM (2016)
4. Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger **4**(2), 29–30 (1993)
5. Fernández-Sánchez, C., Garbajosa, J., Yagüe, A., Perez, J.: Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. J. Syst. Softw. **124**, 22–38 (2017)
6. Highsmith, J.: Agile Project Management: Creating Innovative Products. Pearson Education, London (2009)
7. Huang, Q., Shihab, E., Xia, X., Lo, D., Li, S.: Identifying self-admitted technical debt in open source projects using text mining. Empirical Softw. Eng. **23**(1), 418–451 (2017). https://doi.org/10.1007/s10664-017-9522-4
8. Khadka, R., et al.: How do professionals perceive legacy systems and software modernization? In: Proceedings of the 36th International Conference on Software Engineering, pp. 36–47. ACM/IEEE (2014)
9. Kruchten, P., Nord, R., Ozkaya, I.: Technical debt: from metaphor to theory and practice. IEEE Softw. **29**(6), 18–21 (2012)
10. Kruchten, P., Nord, R., Ozkaya, I.: Technical Debt: Reducing Friction in Software Development. Addison-Wesley, Boston (2019)
11. Lehman, M.M., Belady, L.A.: Program Evolution: Processes of Software Change. Academic Press Professional Inc., Cambridge (1985)
12. Letouzey, J., Ilkiewicz, M.: Managing technical debt with the SQALE method. IEEE Softw. **29**(6), 44–51 (2012)
13. Li, Z., Avgeriou, P., Liang, P.: A systematic mapping study on technical debt and its management. J. Syst. Softw. **101**, 193–220 (2015)

14. Li, Z., Liang, P., Avgeriou, P.: Architectural technical debt identification based on architecture decisions and change scenarios. In: 2015 12th Working IEEE/IFIP Conference on Software Architecture (WICSA), pp. 65–74. IEEE (2015)
15. Parnas, D.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering, pp. 279–287. ACM/IEEE (1994)
16. Russo, D., Ciancarini, P., Falasconi, T., Tomasi, M.: Software quality concerns in the Italian bank sector: the emergence of a meta-quality dimension. In: Proceedings of the 39th International Conference on Software Engineering, pp. 63–72. ACM/IEEE (2017)
17. Russo, D., Ciancarini, P., Falasconi, T., Tomasi, M.: A meta model for information systems quality: a mixed-study of the financial sector. ACM Trans. Manag. Inf. Syst. **9**(3), 1–38 (2018)
18. Tempero, E., Gorschek, T., Angelis, L.: Barriers to refactoring. Commun. ACM **60**(10), 54–61 (2017)
19. Wheelen, T.L., Hunger, J.D.: Strategic Management and Business Policy. Pearson, London (2017)
20. Yli-Huumo, J., Maglyas, A., Smolander, K.: How do software development teams manage technical debt?-An empirical study. J. Syst. Softw. **120**, 195–218 (2016)