Challenges in understanding, measuring, and managing technical debt.

University of Oulu
Information Processing Science
Bachelor Thesis
Sami Wickström
2023

Abstract

Technical debt as a concept is vast and somewhat abstract area. The most basic definition of it is that it is analogous to a bank loan, which must be paid back so you don't get stated bankrupt. Number of activities required to manage the debt are numerous, as 15 different sources of possible debt are identified.

Technical debt management activities, possible sources, and outcomes that unpaid debt might bring are documented in this paper's background section. Results section contains challenges that technical debt management processes are encountering in understanding, measuring, and managing the debt.

Even with existing empirical research on technical debt management, research and industry are having hard time in trying to find the right tools and areas to measure to gain meaningful information on numerous types of technical debt. Without the right metrics, the monitoring tools are not able to provide useful information to aid in communication and decision-making. Currently, the tools focus mainly on code smells (code debt) and are not able to measure the most important aspects of the debt, design, and architectural debt. This research suggests future research topics to improve existing knowledge on certain areas such as the previously mentioned ability to meaningfully measure different kinds of debts.

Keywords
technical debt, TD, technical debt management, TDM

Supervisor
Doctoral Researcher, Leevi Rantala.

Foreword

The vastness of technical debt landscape took me by a surprise. Somewhat infant processes indicate that this is still a worthy research area. This literature review is barely scraping the surface.

Special thanks to the supervisor Leevi Rantala. Feedback has been numerous and helpful.

Contents

1. Introduction

The research towards trying to understand and manage the debt has increased during this last decade to help software stakeholders to manage their technical debt. Mostly theoretical groundwork has been laid out and using that groundwork to manage technical debt in real life has proved to be quite a challenge. Notably because the tools are not sophisticated enough but also because managing the debt is a complex process which requires a lot of resources and competence. With sophisticated enough tools and practices, which are required to be used across whole organization, you could build intelligent tools that automatically calculate the amount of debt and its impact and would make decisions on how to manage the debt, for example decisions about which modules to refactor or reengineer or which processes should be made to work better. (Yli-Huumo et. al., 2016)

This research aims to present up-to-date information on what are the challenges in trying to understand, measure, and manage the debt. It also explains the nature of the debt, how there will always be some amount of technical debt, and why it should be managed (Suryanarayana et. al., 2015).

2. Background

Technical debt is a metaphor first used by Cunningham (1992). He states that it is analogous to financial debt. Meaning that when for example a person takes on a loan and pays it back, he is indeed paying the loan back thus not letting the debt gather and create payment problems. But if the person does not pay the loan back, he or she will be getting a punishment such as a growing loan interest. If the person is not able to pay back the whole loan he or she will be getting stated as bankrupt, which in turn is analogous to gathering too much technical debt to make things right anymore. (Cunningham, 1992) Kruchten et. al (2012) mention that the term itself has been refined multiple times. Now not only being about the source code smells itself but to include other areas of software development lifecycle as well (Kruchten et. al., 2012). One more refinement from 2016 (Avgeriou et. al., 2016) describes the term as sub-optimal design or implementation solutions that might yield benefit in the short term but make changes more costly or even impossible in the medium to long term. Suryanarayana et. al. (2015) says that technical debt is said to start accumulating when quick fix or a shortcut is used and not fixed in-time or at all. Like a bank loan, longer the time it takes to fix the introduced technical debt, the harder it gets to fix it at all as more changes have been introduced. In cases where too much technical debt has gathered and it is no more possible to fix it, this state is said to be technical bankruptcy. (Suryanarayana et. al., 2015 p. 2)

Suryanarayana et. al. (2015) argue that this still does not describe the nature behind the term technical debt fully as there are multiple ways to accumulate technical debt. Some examples of the possible sources of technical debt are code debt, design debt, test debt and documentation debt. *Code debt* refers to static analysis tools violations and using inconsistent coding style or format. *Design debt* refers to lack of well-thought design or violation of the design rules. *Test debt* refers to lack of proper testing. *Documentation debt* refers to lack of documentation, documentation written in a hurry or documentation that does not reflect current state of the system. (Suryanarayana et. al., 2015 pp. 2-4) Zengyang et. al. (2015) identifies more main types of debt: *requirements debt, architectural debt, build debt, infrastructure debt, versioning debt,* and *defect debt.* Rios et. al. (2018) continues the list by identifying 5 more types of debt: *people debt, process debt, automation debt, usability debt,* and *service debt.* Overall, 15 types of technical debt sources are identified. Architectural choices were seen to contribute most to technical debt in a survey (Ernst et. al., 2015).

Suovuo et. al. (2015) gives one more example of technical debt source, connection to external APIs. By utilizing external APIs in your software, you become prone to external changes which might have negative impact on your software (Suovuo et. al., 2015). This means that decisions whether to use external APIs should be weighted according to the impact their possible breaking might bring.

Technical debt seems to refer to already made decisions regarding software. For example, feature backlog of unimplemented features does not constitute as technical debt. But if these features have design TD or some other type of the debt planned, the debt is just waiting to accrue.

2.1 Impact unpaid technical debt might bring

Major refactoring or reengineering can become very costly and if not properly included in business calculations, a surprise might be crushing. AlOmar et. al. (2020) conducted research on who is most likely going to do the refactoring. They analysed 800 open-source projects and came to the same conclusion as others earlier, the ones most likely going to do the refactoring are highly experienced people in high positions with good understanding of the system architecture. (AlOmar et. al., 2020) This supports the idea that quick fixes or other sources of technical debt should be avoided when possible, so there is no need to perform lots of costly refactoring afterwards.

Even acquiring technical debt can have some short-term benefits, the long-term impacts are more negative. Four main outcomes identified (Tom et. al., 2013) are shown in table 1.

Impact	**Explanation**
Morale	Negative effect in long term on morale if the codebase is of not best possible quality.
Productivity	Implementing new features might require major refactoring or reengineering.
Quality	Underlying defects that affect product quality.
Risk	Uncertainty regarding future changes.

Table 1. Four outcomes of unpaid technical debt.

Buschmann (2011) states that the business comes first when handling technical debt. He reasons that there is a natural lifecycle for a product, which means that unpaid technical debt will disappear. He continues that even it might be beneficial to pay the debt, it does not always bring the most value.

2.2 Activities involved trying to manage the debt.

So far, we know about some of the possible sources, outcomes, different forms of technical debt, and it seems that there will always be some amount of debt. A balancing act is needed. In this chapter we go through some activities that needs to be performed to handle the debt.

Yli-Huumo et. al. (2016) observed and interviewed 8 different software teams and 25 people, asking about their technical debt management and use of technical debt management tools. They were able to identify three major stakeholders, developers, software architects, and project managers. Business stakeholder was identified as an additional stakeholder. They found out that development teams usually were involved in repayment, prevention, documentation, and identification during actual development time. Whereas architect was involved in all the TDM activities and was the one keeping it all together. Team manager was observed to be mostly involved in prioritization, communication, monitoring, and measurement. The additional stakeholder, business stakeholder, was involved in communicating with the project manager. It was also quite a welcome surprise that the activity most performed was communication and technical debt was important topic for the teams. Communication gap between the business stakeholder and project manager had tightened. Good communication is the most

important thing to get well working TDM activities up and running smoothly. (Yli-Huumo et. al., 2016)

Still, Yli-Huumo et. al. (2016) continues about occasionally used activities, that while the teams could refactor thus pay the debt to some degree, the most important viewed activity to ease the overall technical debt, prevention, was lacking. There were protocols such as code reviews and predefined coding standards in place, but they were seldomly used as they were not mandatory. Prevention alleviates all other aspects of TDM, so it is important to pay close attention to it. Reasons for the lack of prevention activities were proposed by the researchers. One, it might feel too strict to work around tight guidelines and not let your own creativity shine. Two, it simply requires resources and competence to work this way. Other aspects lacking were TD documentation, simply because it was not thought to be necessary by the teams. The researchers argue that this is not the case and TD documentation is important to improve overall TDM strategy. Same thing with identification, the teams did not feel like they were able to realize what constitutes as technical debt. Then again with prioritization, the teams felt lost trying to prioritize technical debt and just used their hunch and previous knowledge. (Yli-Huumo et. al., 2016)

Zengyang et. al. (2015) in their systematic mapping study examines managing technical debt, especially by using tools specifically created for this purpose called technical debt management tools (TDM). The tools include a set of activities aimed to prevent technical debt from incurring or to at least keep it low enough, so it does not result in technical bankruptcy or big and costly refactoring or reengineering (Zengyang et. al., 2015).

The following Table 2 lists activities of technical debt management tools (TDM) can provide (Zengyang et. al., 2015).

Activity	Understanding	Measurement	Managing
TD identification	Detect technical debt using specific techniques such as static code analysis		
TD measurement		Quantify the benefit and cost of overall technical debt in a system by using estimation techniques	
TD prioritization			Predefined rules to rank current technical debt. Helps in decision-making, which technical debt to fix first and which can be corrected later or completely ignored
TD prevention	Prevent future TD from incurring		
TD monitoring		Measure the cost and benefit of TD over time	
TD repayment			Pay the debt by refactoring or reengineering
TD representation/documentation	Present TD in a unified manner		
TD communication	Use documented TD to communicate with the stakeholders		Use documented TD to help in managing it

Table 2. List of activities TDM tools can provide (Zengyan et. al., 2015).

Zengyang et. al. (2015) continues that of these activities, the most researched activity in selected studies is repayment. Repayment meaning, refactoring, and reengineering. The activity concerned the least is representation and technical debt prevention and communication are third and second last in the presented studies list (Zengyang et al., 2015). These activities seem to relate mainly to understanding and partly to managing the debt. What makes this alarming is the fact that business staff with their lacking technical understanding of the debt are the ones who make important business decisions concerning the organization and its current and future directions.

3. Research methodology

A literature review was chosen as the research method. Some of the guidelines – such as inclusion/exclusion process, from Kitchenham's (2004) systematic literature review guide were used to help formulating a research protocol.

The main points to perform literature review are planning the research, conducting the research, and reporting the results. This process, performed in a somewhat iterative manner, is explained next.

3.1 Planning the research

At first, during the planning stage, it was time to identify a worthy research topic. This was an easy task, and the topic choice was not picked at random as the subject had been found months ago when technical debt as a term first came up. First round of reading documents was performed back then, and it immediately became evident that this topic is worthy of additional research as understanding, managing, and measuring technical debt is a complex subject that has big impact on software domain.

First part for this research was performed last year, few months after the initial reading round. The research question back then was trying to find about problems faced in managing the debt. The used search criteria were same as presented in this section, except the search string was "managing AND software AND technical AND debt". The information acquired answers the same research question presented as re-formatted in the next paragraph. For this reason, this research builds on top of the previous research.

Research question was reformatted to include "understanding", and "measuring" to gain more broader information around the subject and to provide more value. The research question aims to gain deeper understanding of different parts and challenges in play that make understanding, managing, and measuring software technical debt difficult. The updated research question is *"What are the challenges in understanding, managing, and measuring software technical debt?"*

Database to perform the searches on was initially chosen to be Scopus, an abstract and indexing database consisting of peer reviewed research documents.

3.2 Conducting the research

The first step was to reformulate search keywords to match the updated research question better. Visible keywords were readily available from the research question. By combining Boolean operators, a search string was formed. "Understand* OR manag* OR measur* AND software AND technical AND debt".

From the first round of reading about the topic, I was able to remember that research around this topic was said to had accelerated during the last decade. The search results were filtered to include results from 2010-2023 to find up-to-date information, presumed to be more applicable to modern software engineering practices.

From Scopus, by using these criteria, 683 documents were found. This was more than sufficient for the scope of this research, so using only Scopus to search for documents

was decided to be sufficient. By focusing on peer reviewed studies, the data was most likely to be valid and of good quality. The results were sorted by most cited. 10 most cited documents were taken under analysis per round by reading their titles and abstracts to see if they would be fit for this research and provide enough information for the scope of this research. The inclusion process was iterated later until enough relevant information for the scope of this research was found. The aim was to keep the research as bias-free as possible by using most-cited research available as primary search results, thus compiling, and deducing information from most agreed and acknowledged documents. The inclusion criteria were simply “does provide usable information based on title/abstract”, and the exclusion criteria “does not provide usable information based on title/abstract”.

At some point during the study selection iteration process, it became evident that to gain up-to-date information, some more limiting was necessary. The search results were updated to include results only from 2018–2023 (February). Other documents referenced in found documents were also used if they provided valuable information. As well as one document suggested by the supervisor of this research, who has expertise of the domain.

Finally, the findings were thoroughly analysed, with an aim to find and compile up-to-date information answering the multi-faceted research question and for further research. These were documented to results section and further discussed in discussion section.

4. Results

Here the research results are presented regarding the multi-faceted research question: what are the challenges in *understanding, managing*, and *measuring* software technical debt, grouped under their respective headings.

4.1 Understanding technical debt

Technical debt is said to be observed mainly during maintenance and can catch you by surprise (Avgeriou et. al., 2016 p. 111). Different set of activities are needed to methodically track and understand the various types of the debt (Zengyan et. al., 2015).

In one study by Kaiser and Royse (2011) the researchers conducted an experiment to help a company better understand their technical debt. They fed to CSV format transformed source code data into Microsoft Tree Mapper and printed four-foot-tall diagrams of the codebase and placed them in the hallway between two teams working on the same source code. The visuals represented source code's cyclomatic complexity, meaning the number of linearly independent paths through a source code. Red and green, the colours often used to symbolise good and evil, were used to colour the tree representation by potential areas of trouble. It turned out that this sparked interest between the stakeholders and helped them to understand technical debt they were taking on. IT-directors of said company kept the tree representations visible even afterwards as they thought it brought significant value. (Kaiser & Royse, 2011)

The tools for managing technical debt aim to help more business-minded people to also understand the impact that too much unpaid technical debt can bring (Zengyang et. al., 2015, pp. 204-205). Problems faced in trying to understand the debt are that TD monitoring was seen to be very minimal as there were not much to monitor because TD measurements did not bring any true value (Yli-Huumo et. al., 2016, p. 212).

4.2 Measuring technical debt

There are different levels of technical debt to measure, and measuring the code debt is only one part of the equation. There is a need to be able to measure other previously identified types of technical debt and their impact as well. Fitting tools and measures are said to be somewhat missing, and developing such tooling is a challenge for research and industry (Avgeriou et. al., 2016; Ernst et. al., 2015). Developers feel they don't have enough resources to measure and to communicate about the debt and its impact to managers who are clueless about the amount of debt and the value that managing it could bring (Ernst et. al., 2015).

Yli-Huumo et. al. (2016) were able to identify rarely used TD management activities. TD measurement was simplistic and did not offer much insight. The reason mentioned for this is that there are simply no good enough tools to measure right areas, such as major architectural flaws. Simply monitoring code quality is not enough to spot these design trade-offs that are said to be contributing most to technical debt (Avgeriou et. al., 2016). Fontana et. al. (2016, as cited in Besker et. al., 2018) researched TD indexing tools to find out which tools took architectural debt into account and found out that no tools provided good results and did not help to better understand the amount of architectural debt in a project.

One novel approach to measuring technical debt is suggested by Liu et. al. (2018), a text-mining tool that scans source code comments for self-admitted technical debt (SATD) written by developers.

In a more up-to-date study, a group of researchers (Avgeriou et. al., 2021) analysed 9 different technical debt management tools and their performance. Things to analyse were type, principal, interest, and index. These tools are all aimed to analyse source code, so they are not fit for measuring every aspect of technical debt. The research suggests that certain tools might be better in terms of extra features such as displaying interest (consequence) and architectural debt in addition to just debt from code smells. Static analysis is only used in most of the tools to calculate technical debt index, but it is suggested that more information sources such as version history, issue tracker, emails, could be included for more precise calculations. The rules and metrics on how the debt is measured also differ widely, and it could be useful to provide more uniform metrics. The research also acknowledges that the tools are still missing vital means to calculate architectural debt. (Avgeriou et. al., 2021)

4.3 Managing technical debt

It should be noted that there will always be some amount of technical debt and that the outcome of accrued debt is not always negative (Tom et. al., 2013). Decisions whether to refactor can be hard to make as the benefits can be vague and seen in long term whereas the costs are visible immediately (Kazman et. al., 2015).

Martini et. al. (2018) realized that even quite a lot of theoretical work has been conducted regarding technical debt, empirical research is still lacking. They conducted comprehensive research on how 15 large organizations manage their technical debt. The results were quite alarming. They found out that of the overall development time, 25% is used to manage the debt. This percent might seem promising at first, but they also found out that it was mostly not systematic in nature. 26% of the participants used a technical debt management tool, and only 7.2% of the participants tracked the technical debt methodically. Tools that were mostly used were simply backlogs and static code analysis tools. Participants also reported that it was hard to get acceptance from the IT managers for refactoring. When risk/impact value is not kept track of, it is hard to make decisions. They also reported that by simply making the technical debt visible in the technical debt backlog, was not convincing reason for business staff to accept refactoring as the metrics were not tracked. (Martini et. al., 2018)

Martini et. al. (2018) suggests companies to adopt tools and practices often mentioned in theoretical research. They identified three steps which would lead to automated data-driven technical debt management. First step, called measured, is to start using the tools to gather metrics that would help in decision making. Next step, called institutionalized, is to make these tools and practises available and used by whole organization. The last step, called fully automated, is to fully automate the decision-making process by utilizing the data these tools and practices provide. (Martini et. al., 2018)

A team of researchers (Yli-Huumo et. al., 2016) were able to compile the biggest challenges in technical debt management (TDM). One, lack of proper tools. Two, knowledge of priorities as it requires insight from other areas of TDM to work. Three, wrong mindset. Four, managing TDM requires resources and is time-consuming. (Yli-Huumo et. al., 2016)

Initial architectural choices such as opting for micro services, opposed to monolithic architecture, can limit the reach that technical debt has over maintainability (Avgeriou et. al., 2016).

5. Discussion

Understanding the consequences that taking on debt might bring is an important topic. Consequences that can manifest in maintainability should be well understood to support decision-making among stakeholders, so that it does not take you by surprise during maintenance. Without right areas to measure, the debt monitoring does not bring any true value to better understanding the debt and consequences it can bring in the long run.
It is not entirely clear besides visual representation how the debt should be represented so that business-minded people could understand the consequences that unmanaged debt can bring. It was said that even if the debt was made visible in the backlog, the decisions whether to pay the debt were hard to make and were not often justified by the decision-makers without the actual metrics for long-term maintainability consequences and lacking full-spectrum view of the debt.

Measuring different areas besides code smells should be taken care of, as there are multiple possible sources of TD. Measurements should target the right metrics and provide prioritization over these metrics, so that the monitoring can be seen useful.
Visual representation of code smells seems to be a good driver to understand code-based technical debt and that much is offered by the usual tools. Lacking is the ability to efficiently monitor the most important aspects of the debt, architectural and design debt. Self-admitted technical debt reporting in the form of developer comments is suggested as one possible way to provide measurements of architectural debt, but this requires that the developers are willing to admit the use of suboptimal solutions. It is also quite problematic that there are no definite guidelines on how the different debt indexes should be calculated, and this creates problem in deciding which tool to use and how it compares to some other tools.

Managing the debt in an automated data-driven way is the aim but such tools are still a long way to go. Systematic management is hard due to missing guidelines of best practices and ways to measure the right areas in meaningful standards. It is not clear how the debt should be methodically managed, and the lack of general measures makes decision-making processes problematic.
As some guidelines to present, the process of managing the debt should be started early-on in the development-cycle and it should be kept in mind while opting for architecture, as some architectures support managing the debt better than others. Development processes with differing activities play a role in here as well as refactoring and reengineering is strongly considered paying the debt.
Empirical research is there but it is lacking in certain important aspects such as prioritization of the sources of debt. Validation of these aspects is a problem faced in trying to develop the right metrics. Naturally, the lack of studies on a certain domain mirror to real life, this meaning that these are some of the negated parts of managing the technical debt. It was mentioned in multiple studies how the consequences that certain architectural design trade-offs will bring should be studied to further the understanding of architectural debt.

6. Conclusion

Understanding the debt is the first part in the process of being able to weight the pros and cons of debt methodically. Problems faced with proper understanding of the debt relate to missing tools that provide meaningful metrics that can help in decision-making processes. The tools should be able to measure all identified 15 types of TD sources and provide prioritization over these types and their repayment so they could be seen useful in communication and as decision-making tools. The term technical debt is an important one in communication about the consequences the types of debt can bring among stakeholders. Adding more debt can make the business-minded people happy, and developers grumpy because of the introduced design trade-offs. The definition implies that the debt is often incurred strategically but it does not mean that the consequences are fully understood at the time.

It is hard to develop the right metrics so that all 15 identified types of TD could be measured in a meaningful way that provides value to communication and decision-making. Currently the measurements relate mainly to code smells (code debt) and neglect the rest of the types. Even these measurements vary between the tools and there are no clear guidelines how the metrics should be calculated and represented. The lack of proper metrics is a challenge in creating future management tools. Rios et. al. (2018) in their tertiary study on current state of TD research describes what kind of research topics have been investigated regarding TD as well as future directions. The results described reflect this document's purpose, as the most researched topics are TD concepts, identification, and management. Future research directions identified consists of better understanding other types of TD than code debt, how to use indicators of other types of debt effectively in identifying TD, and scenario-based decision-criteria usable in planning to manage the debt payment. (Rios et. al., 2018)

The debt should be managed throughout a whole product lifecycle so that optimal solutions can be opted for in time, if felt necessary, before they can affect maintainability. Fitting holistic tools for managing the debt are missing and current tools do not provide meaningful measures. Different set of activities required to manage the debt organization-wide are presented in this document. For future research, Rios et. al. (2018) presents an exhaustive TD management landscape, consisting of activities, strategies, and tools used to manage these activities. As imaginable, by now knowing the dimensions of technical debt as a concept, the landscape consists of tens of different tools and strategies suited for different activities. The identified activities include roughly the same activities discussed earlier in chapter 2.2 (Zengyang et. al., 2015). At this point, a single solution to manage the whole technical debt spectrum of identified 15 different sources of technical debt seems like scientific fiction.

References

AlOmar, E. A., Peruma, A., Newman, C. D., Mkaouer, M. W., & Ouni A. (2020). On the Relationship Between Developer Experience and Refactoring: An Exploratory Study and Preliminary Results. *ICSEW'20: Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*. pp. 342-349. https://doi.org/10.1145/3387940.3392193.

Avgeriou, P. Kruchten, I. Ozkaya, & C. Seaman. (2016). Managing Technical Debt in Software Engineering (Report from Dagstuhl Seminar 16162). *Dagstuhl Reports*, vol. 6, no. 4, pp. 110–138.

Avgeriou, P., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigerorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D., Soares de Toledo, S., & Agathi Tsintzira, A. (2020). An Overview and Comparison of Technical Debt Measurement Tools. *Ieee software, 38(3)*, 61-71. https://doi.org/10.1109/MS.2020.3024958.

Besker, T. Martini, A. Bosch, J. (2018). Managing architectural technical debt: A unified model and systematic literature review. *The Journal of Systems and Software 135.*

Buschmann, F. (2011). To pay or Not to pay Technical debt. *IEEE Software*, 20(6).

Cunningham, W. (1992). The WyCash portfolio management system. *Addendum to the proceedings on object-oriented programming systems, languages, and applications.* OOPSLA '92; 1992.

Ernst, N., A., Bellomo, S., Ozkaya, I., Nord, R., L., Gorton, I. (2015). Measure it? Manage it? Ignore it? software practitioners and technical debt. *Proceedings of the 2015 10th Joint meeting on Foundations of Software Engineering*, 50–60.

Iu, Z. Huang, Q. Xia, X. Shibah, E. Lo, D. & Li, S. (2018). SATD detector: A text-mining-based self-admitted technical debt detection tool. *ICSE 2018: Proceedings of the 40th ACM/IEEE International Conference on Software Engineering: Gothenburg, Sweden, May 27 - June 3. 9-12.*

Kaiser, M., Royse, G. (2011). Selling the Investment to Pay Down Technical Debt. *Proceedings - 2011 Agile Conference*, Agile 2011, pp. 175-180.

Kazman, R., Cai, Y., Mo, R., Feng, Q., Xiao, L., Haziyev, S., Fedak, V. & Shapochka, A. (2015). A Case Study in Locating the Architectural Roots of Technical Debt. *IEEE/ACM 37th IEEE International Conference on Software Engineering*, Florence, Italy, pp. 179–188, https://ieeexplore.ieee.org/document/7202962.

Kitchenham, B., (2004). *Procedures for Undertaking Systematic Reviews*. Joint Technical Report, Computer Science Department, Keele University (TR/SE-0401) and National ICT Australia Ltd (0400011T.1).

Kruchten, P., Nord, R. L. & Ozkaya, I. (2012). Technical Debt: From Metaphor to Theory and Practice. *IEEE Software*, 29(6).

Martini, A., Besker, T., & Bosch, J. (2018) Technical Debt tracking: Current state of practice A survey and multiple case study in 15 large organizations. *Science of Computer Programming,* 163, pp. 42-61.

Rios, N., Mendonça Neto, M.G.D., Spínola, R.O. (2018). A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102, 117-145.

Suovuo, T., Holvitie, J., Smed, J., & Leppänen, V. (2015). Mining knowledge on technical debt propagation. *CEUR Workshop Proceedings* 1525, pp. 281-295. http://ceur-ws.org/Vol-1525/paper-20.pdf.

Suryanarayana, G., Samarthyam, G., & Sharma, T. (2015). *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann. https://doi.org/10.1016/B978-0-12-801397-7.00001-1.

Tom, E., Aurum, A. & Vidgen, R. (2013). An exploration of technical debt. *Journal of Systems and Software*, 86(6), pp. 1498-1516.

Yli-Huumo, J., Maglyas, A., & Smolander, K. (2016). How do software development teams manage technical debt? - An empirical study. *The Journal of Systems and Software* 120, pp. 195-218.

Zengyang, L., Paris, A. & Peng, L. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, 101, pp. 193-220. https://doi.org/10.1016/j.jss.2014.12.027.