THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

Technical Debt: An empirical investigation of its harmfulness and on management strategies in industry

TERESE BESKER

DIVISION OF SOFTWARE ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
CHALMERS UNIVERSITY OF TECHNOLOGY
GOTHENBURG, SWEDEN 2020

المنارة للاستشارات

www.r

ProQuest Number: 28193674

All rights reserved

INFORMATION TO ALL USERS
The quality of this reproduction is dependent on the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript
and there are missing pages, these will be noted. Also, if material had to be removed,
a note will indicate the deletion.

ProQuest 28193674

Published by ProQuest LLC (2020). Copyright of the Dissertation is held by the Author.

All Rights Reserved.
This work is protected against unauthorized copying under Title 17, United States Code
Microform Edition © ProQuest LLC.

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

المنارة للاستشارات

www.manaraa.com

Technical Debt: An empirical investigation of its harmfulness and on management strategies in industry

TERESE BESKER

Doktorsavhandlingar vid Chalmers tekniska högskola
ISBN: 978-91-7905-274-4
Series number: 4741
ISSN: 0346-718X

Technical Report No 182D
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Copyright ©2020 Terese Besker
except where otherwise stated.
All rights reserved.

Printed by Chalmers Reproservice,
Göteborg, Sweden 2020.

المنارة للاستشارات

www.r

"The first step is to establish that something is possible; then probability will occur."

Elon Musk

المنارة للاستشارات

www.r

المنارة للاستشارات

www.m

Abstract

Background: In order to survive in today's fast-growing and ever fast-changing business environment, software companies need to continuously deliver customer value, both from a short- and long-term perspective. However, the consequences of potential long-term and far-reaching negative effects of shortcuts and quick fixes made during the software development lifecycle, described as Technical Debt (TD), can impede the software development process.

Objective: The overarching goal of this Ph.D. thesis is twofold. The first goal is to empirically study and understand *in what way* and *to what extent*, TD influences today's software development work, specifically with the intention to provide more quantitative insight into the field. Second, to understand which different initiatives can reduce the negative effects of TD and also which factors are important to consider when implementing such initiatives.

Method: To achieve the objectives, a combination of both quantitative and qualitative research methodologies are used, including interviews, surveys, a systematic literature review, a longitudinal study, analysis of documents, correlation analysis, and statistical tests. In seven of the eleven studies included in this Ph.D. thesis, a combination of multiple research methods are used to achieve high validity.

Results: We present results showing that software suffering from TD will cause various negative effects on both the software and the developing process. These negative effects are illustrated from a technical, financial, and a developer's working situational perspective. These studies also identify several initiatives that can be undertaken in order to reduce the negative effects of TD.

Conclusion: The results show that software developers report that they waste 23% of their working time due to experiencing TD and that TD required them to perform additional time-consuming work activities. This study also shows that, compared to all types of TD, *architectural* TD has the greatest negative impact on daily software development work and that TD has negative effects on several different software quality attributes. Further, the results show that TD reduces developer morale. Moreover, the findings show that intentionally introducing TD in startup companies can allow the startups to cut development time, enabling faster feedback and increased revenue, preserve resources, and decrease risk and thereby contribute to beneficial effects. This study also identifies several initiatives that can be undertaken in order to reduce the negative effects of TD, such as the introduction of a tracking process where the TD items are introduced in an official backlog. The finding also indicates that there is an unfulfilled potential regarding how managers can influence the manner in which software practitioners address TD.

Keywords: *Software Engineering, Technical Debt, Software Architecture, Software Quality, Software Developing Productivity, Developer Morale, Empirical Research, Mixed-methods*

المنارة للاستشارات

www.r

المنارة للاستشارات

www.m

Acknowledgments

First of all, I would like to express my deepest gratitude and appreciation to my main supervisor, Professor Jan Bosch, for his encouragement, support, guidance, and engagement. You continuously raise the bar with me, and, most importantly, make me believe I can reach my goals.

Next, I would like to express my sincere appreciation to my second supervisor, Professor Antonio Martini, for always sharing his technical knowledge and expertise. Besides being a great friend, your support, ideas, and comments have significantly improved the quality of my research.

I would also like to thank my colleagues David Issa Mattos and Magnus Ågren, for fruitful discussions and great friendship.

I want to thank all the partners at the Software Center for supporting my research and ensuring that we conduct research into highly relevant topics from both an academic and an industrial software perspective.

Finally, I would like to express my sincere gratitude to my family and friends for their love and continuous encouragement.

Terese Besker

Gothenburg, Sweden, 2020

المنارة للاستشارات

www.r

المنارة للاستشارات
www.m

List of Publications

Appended Papers

This Ph.D. thesis is based on the work contained in the following peer-reviewed publications:

[A] **T. Besker**, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software, vol. 135, pp. 1–16, 2018.*

[B] **T. Besker**, A. Martini, and J. Bosch, "Time to Pay Up – Technical Debt from a Software Quality Perspective," *In proceedings of the 20th Ibero American Conference on Software Engineering (CibSE) @ ICSE17, 2017.*

[C] **T. Besker**, A. Martini, and J. Bosch, "The Pricey Bill of Technical Debt – When and by Whom Will it be Paid?" *Proceedings of IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, pp. 13–23, 2017.*

[D] **T. Besker**, A. Martini, and J. Bosch, "Impact of Architectural Technical Debt on Daily Software Development Work – A Survey of Software Practitioners," *Proceedings in 43th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vienna, 2017, pp. 278–287.*

[E] **T. Besker**, A. Martini, and J. Bosch, "Software developer productivity loss due to technical debt – A replication and extension study examining developers' development work," *Journal of Systems and Software, vol. 156, pp. 41–61, 2019.*

[F] **T. Besker**, H. Ghanbari, A. Martini, and J. Bosch, "The Influence of Technical Debt on Software Developer Morale," *Journal of Systems and Software, vol. 167, pp. 110586, 2020.*

[G] A. Martini, **T. Besker**, and J. Bosch, "Technical Debt Tracking: Current State of Practice – A Survey and Multiple Case-Study in 15 Large Organizations," *Science of Computer Programming, 2017.*

[H] **T. Besker**, A. Martini, R. E. Lokuge, K. Blincoe, and J. Bosch, "Embracing Technical Debt, from a Startup Company Perspective," *IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 415–425, 2018.*

المنارة للاستشارات

www.r

[I] **T. Besker**, A. Martini, and J. Bosch, “How Regulations of Safety-Critical Software Affect Technical Debt,” *45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 74–81, 2019*.

[J] **T. Besker**, A. Martini, and J. Bosch, “Technical debt triage in backlog management,” *Proceedings of the Second International Conference on Technical Debt, Montreal, Quebec, Canada, 2019, pp. 13–22, 2019.*

[K] **T. Besker**, A. Martini, and J. Bosch, “Carrot and Stick approaches when managing Technical Debt,” *Proceedings of the Third International Conference on Technical Debt, co-located with ICSE, South Korea, 2020. In print.* This paper was granted the ***Best Paper Award*.**

المنارة للاستشارات

www.m

Other Publications

The following publications are peer-reviewed and published but not appended to this thesis:

[L] **T. Besker**, A. Martini, and J. Bosch, "A Systematic Literature Review and a Unified Model of ATD," *Proceedings in 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Cyprus, 2016, pp. 189–197.*

[M] **T. Besker**, A. Martini, and J. Bosch, "Technical Debt Cripples Software Developer Productivity – A longitudinal study on developers' daily software development work," *Proceedings in First International Conference on Technical Debt @ ICSE18, 2018.*

[N] A. Martini, **T. Besker**, and J. Bosch, "The introduction of Technical Debt Tracking in Large Companies," *Proceedings in the 23rd Asia-Pacific Software Engineering Conference (APSEC), Hamilton, New Zealand, 2017*.

[O] **T. Besker**, A. Martini, J. Bosch, and M. Tichy, "An investigation of technical debt in automatic production systems," *Proceedings of the XP2017 Scientific Workshops, Cologne, Germany, 2017.*

[P] H. Ghanbari, **T. Besker**, A. Martini, and J. Bosch, "Looking for Peace of Mind? Manage your (Technical) Debt – An Exploratory Field Study," *Proceedings in the International Symposium on Empirical Software Engineering and Measurement (ESEM), Toronto, Canada, 2017*

[Q] P. Avgeriou, D. Taibi, A. Ampatzoglou, F. A. Fontana, **T. Besker,** A. Chatzigeorgiou, V. Lenarduzzi, A. Martini, N. Moschou, I. Pigazzini, N. Saarimäki, D. Sas, S. Soares de Toledo, and A. Tsintzira, "An Overview and Comparison of Technical Debt Measurement Tools," *Revised version submitted to IEEE Software Journal, 2020*

[R] V. Lenarduzzi, **T. Besker**, D. Taibi, A. Martini, and F. Francesca Arcelli, "Technical Debt Prioritization: State of the Art. A Systematic Literature Review," *Revised version submitted to Journal of Systems and Software, 2020.*

www.m

المنارة للاستشارات

www.m

Personal Contribution

For all publications where I am the first author, my contribution is listed below using the CRediT (Contributor Roles Taxonomy) author statement [1], where I made the following contributions:

a) Conceptualization – Formulation of overarching research goals and aims.

b) Methodology – Design of methodology and creation of research models.

c) Validation/Verification – Focus on the overall replication/reproducibility of results.

d) Formal Analysis – Application of statistical techniques to analyze or synthesize data.

e) Investigation – Conducting the research process and performing data collection.

f) Data Curation – Activities to annotate scrub data and maintain research data.

g) Writing – Original draft, review, and editing.

h) Preparation – Creation and/or presentation of the published work.

i) Project Administration – Management and coordination responsibility for research activities.

For the included publication paper G, in which I am listed as a second author, I made the following contributions:

a) Conceptualization – Formulation of overarching research goals and aims.

b) Methodology – Design of methodology and creation of research models.

c) Investigation – Conducting the research process and performing data collection.

e) Data Curation – Activities to annotate scrub data and maintain research data.

e) Writing (partly) – Original draft, review, and editing.

i) Project Administration – Management and coordination responsibility for research activities.

المنارة للاستشارات

www.r

المنارة للاستشارات

www.m

Table of Content

1. Introduction 1

2. Background and Related Work 5

2.1. The concept of Technical Debt 5

2.2. The impact of Technical Debt in the software development industry 9

2.3. Technical Debt in context-specific domains 13

3. Research Motivation 17

4. Research Questions 19

5. Relationship between Studies and Research Questions 20

5.1. Research studies addressing RQ1 20

5.2. Research studies addressing RQ2 21

5.3. Research studies addressing RQ3 21

6. Methodology 23

6.1. Research Approaches 24

7. Data analysis 31

7.1. Quantitative Data Analysis 31

7.2. Qualitative Data Analysis 31

7.3. Threats to Validity 32

8. Overview of Papers and Findings 37

8.1. Paper A: Managing Architectural Technical Debt: A unified model and systematic literature review 37

8.2. Paper B: Time to Pay Up – Technical Debt from a Software Quality Perspective 39

8.3. Paper C: The Pricey Bill of Technical Debt – When and by Whom Will it be Paid? 41

8.4. Paper D: Impact of Architectural Technical Debt on Daily Software Development Work – A Survey of Software Practitioners 42

8.5. Paper E: Software Developer Productivity Loss Due to Technical Debt – A replication and extension study examining developers' development work........ 44

8.6. Paper F: The Influence of Technical Debt on Software Developer Morale........ 45

8.7. Paper G: Technical Debt Management: Current State of Practice........ 46

8.8. Paper H: Embracing Technical Debt, from a Startup Company Perspective........ 47

8.9. Paper I: How Regulations of Safety-Critical Software Affect Technical Debt 48

8.10. Paper J: Technical debt triage in backlog management........ 49

المنارة للاستشارات

www.r

8.11. Paper K: Carrot and Stick approaches when managing Technical Debt............... 49
9. Managing Architectural Technical Debt 51
9.1. Introduction 51
9.2. Background.. 54
9.3. SLR method.. 56
9.4. Results from the retrieval of publications.. 62
9.5. Results 67
9.6. Importance of ATD and a need for a Unified Model............................ 72
9.7. Discussion... 73
9.8. Related work... 78
9.9. Conclusions 79
10. Technical Debt from a Software Quality Perspective 81
10.1. Introduction 81
10.2. Related work... 82
10.3. Methodology... 83
10.4. Results and findings.. 86
10.5. Discussions and Limitations 92
10.6. Threats to validity... 94
10.7. Conclusion.. 94
11. The Pricey Bill of Technical Debt 97
11.1. Introduction 97
11.2. Related work... 99
11.3. Methodology... 100
11.4. Results and Findings... 104
11.5. Discussion... 115
11.6. Threats to Validity and Verifiability.. 116
11.7. Conclusion.. 117
12. Impact of Architectural Technical Debt on Software Development Work .. 119
12.1. Introduction 119
12.2. Related work... 122
12.3. Methodology... 124
12.4. Results and Analysis.. 126
12.5. Discussion and limitations... 134

المنارة للاستشارات

www.r

12.6. Threats to validity and Verifiability 136
12.7. Conclusion 136
13. Software Developer Productivity Loss Due to Technical Debt 139
13.1. Introduction 139
13.2. Research Questions 142
13.3. Related Work 142
13.4. Methodology 144
13.5. Results and Findings 154
13.6. Discussion 171
13.7. Verifiability, limitations, and threats to validity 175
13.8. Conclusion and future work 178
14. The Influence of Technical Debt on Software Developer Morale 181
14.1. Introduction 181
14.2. Theoretical background 184
14.3. Methodology 191
14.4. Results 197
14.5. Discussion 213
14.6. Limitations, and Threats to Validity 217
14.7. Conclusion 219
15. Technical Debt Tracking: Current State of Practice 221
15.1. Introduction 221
15.2. Methodology 223
15.3. Results 230
15.4. Discussion 245
15.5. Conclusion 251
16. Technical Debt, from a Startup Company Perspective 253
16.1. Introduction 253
16.2. Background and related work 254
16.3. Research Methodology 257
16.4. Description of cases 260
16.5. Results 261
16.6. Discussion 268
16.7. Conclusion 273

المنارة للاستشارات

www.r

17. Technical Debt in Safety-Critical Software 275
17.1. Introduction 275
17.2. Related work 277
17.3. Research method 278
17.4. Results and findings 283
17.5. Discussion 288
17.6. Limitations and threats to validity 291
17.7. Conclusion 291
18. Prioritization of Technical Debt in Backlogs 293
18.1. Introduction 293
18.2. Research model and background 294
18.3. Research Methodology 297
18.4. Results and findings 302
18.5. Discussion 309
18.6. Study Limitations 312
18.7. Conclusion 312
19. Carrot and Stick approaches when managing Technical Debt 313
19.1. Introduction 313
19.2. Conceptual framework 315
19.3. Background and related work 316
19.4. Methodology 318
19.5. Results and findings 324
19.6. Discussion and limitations 330
19.7. Implications for future practice 331
19.8. Threats to validity 332
19.9. Conclusion 333
20. Answering the Thesis' Research Questions 335
20.1. RQ1 – Consequences of TD in today's software industry 335
20.2. RQ2 – Initiatives to reduce the negative effects of TD in today's software industry 337
20.3. RQ3 – Factors affecting TD management in context-specific domains 339
21. Future Research 341
21.1. Technical Debt in Machine Learning applications 341

المنارة للاستشارات
www.r

21.2. Process Debt 342
22. Conclusion **343**
23. Bibliography **347**

المنارة للاستشارات

www.r

www.m

1. Introduction

In order to survive in today's fast-growing and ever-changing business environment, large-scale software companies need to continuously deliver customer value, both from a short- and long-term perspective. Today's engineering of software involves several different activities, such as development, design, testing, implementation, and maintenance of software [2]. During the software development lifecycle, companies need to consider the tradeoffs between the time and effort spent on increasing the overall quality of the software, and the costs of the software development process in terms of the required time and resources. In general, software companies strive to balance the quality of their software with the ambition of increasing the efficiency and decreasing the costs in each lifecycle phase, by reducing time and resources deployed by the development teams.

Examples of this tradeoff can be illustrated by scenarios where software companies deliberately implement sub-optimal solutions, such as, e.g., implementation of "quick fixes" or "cutting corners" in order to reduce the development time and thereby shorten the time-to-market, or when they are forced to implement sub-optimal solutions due to the fact that available resources are limited.

Even if the best intention is to go back and refactor the sub-optimal solution immediately afterward, there is a tendency for these refactoring tasks to be postponed since, typically, there are other important deadlines in the near future, and these tasks are thus often down prioritized. There is also the scenario where sub-optimal solutions are implemented unintentionally due to, e.g., lack of knowledge, guidelines, or best practices.

As a result of these scenarios, the sub-optimal solutions in the software gradually grow, and the short-term implemented quick fixes in the code base live on and become more deeply embedded. Last-minute hacks remain in the code and turn into features that the users depend upon, documentation and coding conventions are potentially also ignored, and eventually, the original architecture degrades and becomes obfuscated [3]. When new requirements that necessitate the software to be extended and altered start appearing, these implemented sub-optimal solutions can become costly to refactor and can also cause delays and thereby also impede both innovation and expansion of the software system.

The result of this impediment is the accrual of what is described as Technical Debt (TD). The TD metaphor was first coined at OOPSLA '92 by Ward Cunningham [8], to describe the need to recognize the potential long-term negative effects of immature code implemented during the software development lifecycle. Cunningham used the financial terms "debt" and "interest" when describing the concept of TD: "Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt."

An additional more recent definition of TD was provided by Avgeriou et al. [4], who define TD as "In software-intensive systems, technical debt is a collection of design or

المنارة للاستشارات

www.r

implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability".

An illustration of the above statement – "technical context where the future changes are more costly or impossible" – may be exemplified by a situation where the software experiencing TD becomes fragile in terms of the occurrence of unexpected side-effects or when changes to one part of the software cause unpredicted failures in other unrelated parts. This situation could make software practitioners avoid altering the software actively.

If the sub-optimal solution refers to the architecture of the system, this can be illustrated by a situation where the architecture is inflexible in terms of resistance to changeability. Without first implementing extensive, costly, risky, and time-consuming architectural refactoring, the possibility of implementing new features is significantly reduced. In a worst-case scenario, software companies could reach a point where they have accrued so much TD that they spend more time maintaining and containing their legacy software than adding new features for their customers [3]. Accumulated negative consequences of TD can even lead to a crisis point when a huge, costly refactoring or a replacement of the entire software needs to be undertaken [5].

Even if there are some special situations and circumstances where deliberately taking on TD can be beneficial for companies (e.g., in startup companies [6]), TD is in general considered to be detrimental to the long-term success of software [7], and, left unchecked, can result in compromised quality attributes such as maintainability, reusability, performance, and the ability to add new features. In addition to potential quality complications, TD can also hinder the software development process by causing an excessive amount of waste of working time in terms of low developer productivity, project delays, high defect rates [8] and TD can also cause significant economic losses [9]. Further, several studies suggest that TD also has a negative influence on developers' emotions [10],[11],[12] and on their morale [7],[13].

Although the concept and harmfulness of TD are gaining importance from an academic perspective, software companies still struggle with paying TD management sufficient attention in practice. There are several major reasons for this, such as the difficulty of implementing prevention mechanics to avoid introducing TD in the first place, of raising awareness of the negative effects TD has on the overall software development process, and difficulties in understanding and quantifying the level of negative impact from TD.

Moreover, when TD is present in the software, the only significantly effective way of reducing it is to refactor the software. However, the refactoring activities of the different identified TD items need to be prioritized both individually and also in competition with, for example, the implementation of new features. In this sense, software managers' mindset can also have an impact on the way practitioners address, prioritize, and focus their attention on TD remediation activities.

Furthermore, there are several different types of TD [14],[15],[7], such as, e.g., Architectural TD, Documentation TD, Requirement TD, Code TD, Test TD, and

المنارة للاستشارات

www.r

Infrastructure TD. These different TD types affect different parts of software development during different development phases, and they also have different levels of negative impact on the overall software development process. This Ph.D. thesis focuses on all TD types, even if some of the included publications more specifically focus on the Architectural TD (ATD) type. ATD is often described as the most important source of TD [16] as well as the most frequently encountered type of TD [17].

The overall goal of this Ph.D. thesis is to study and understand *in what way* and *to what extent* TD influences today's software development work from various perspectives, and also to understand what different initiatives can be undertaken to reduce the negative effects of TD.

The remainder of this thesis is structured in 23 chapters: Chapter 2 describes the Background and Related Work, and Chapter 3 introduces the Research Motivation. Chapter 4 describes the Research Questions. Chapter 5 presents the Relationship between Studies and Research Questions. Chapter 6 and 7 discuss the Methodology and the Data analysis, respectively. Chapter 8 presents an Overview of Papers and Findings. Chapter 9 to 19 presents results from the eleven included publications. Chapter 20 presents the Answers to the Thesis' Research Questions. In chapters 21 and 22, future work and conclusions are presented.

The overall structure of this Ph.D. thesis is illustrated in Figure 1.

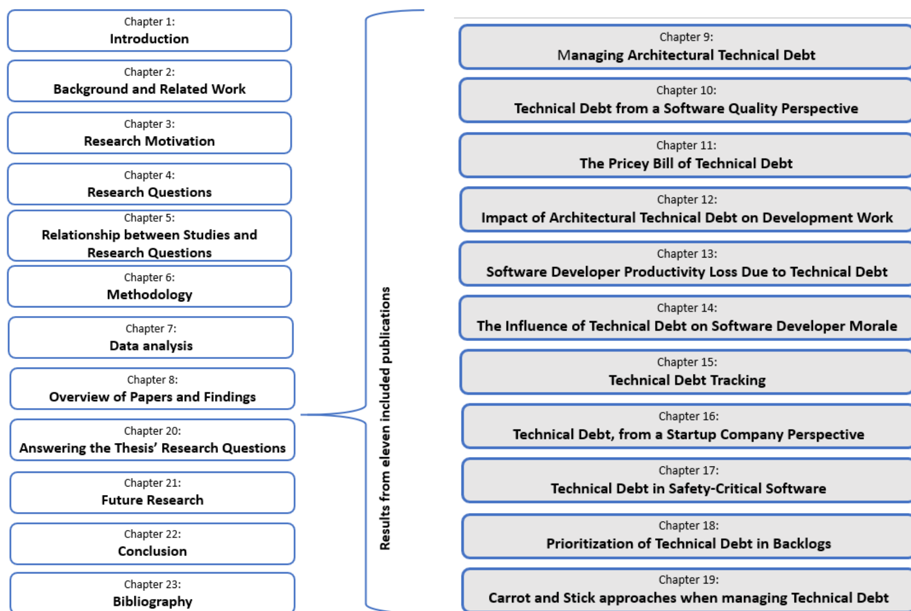


Figure 1: Thesis overview

المنارة للاستشارات

www.r

للاستشارات المنارة
www.m

2. Background and Related Work

This thesis studies TD from different perspectives, and in order to provide the reader with the necessary information needed to better understand the remainder of the thesis, this chapter provides background information and describes the related work of the thesis.

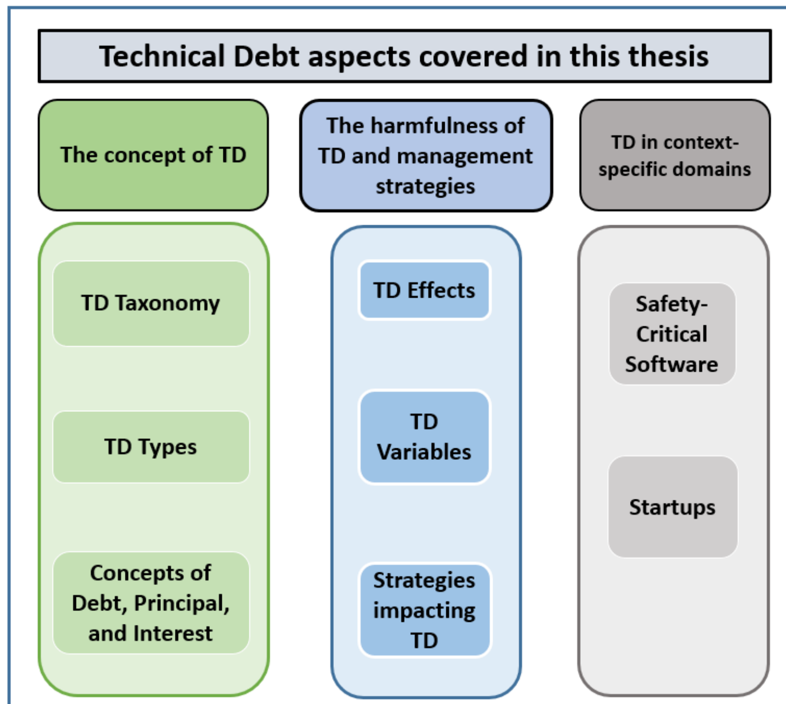


Figure 2: A background overview model of Technical Debt, as portrayed in this Ph.D. thesis.

Figure 2 is an overview model that illustrates the essential aspects of the concerned included research topics, which are all addressed in this Ph.D. thesis. As presented in the figure, the topics are organized using a three-dimensional model where the first dimension addresses the concept of TD, followed by a second dimension where the harmfulness and TD management strategies are in focus, followed by the third dimension which explicitly assesses TD in two different company context-specific domains.

The following sections describe more in detail how each of these dimensions relates to each other and also how the aspects in each dimension relate to each other.

2.1. The concept of Technical Debt

Figure 3 illustrates the covered aspects of the first dimension in the overview Figure 2, "The concept of TD," and how these aspects relate to the other dimensions. The figure shows, for example, that causes of the different investigated TD types are assessed in the

المنارة للاستشارات

www.r

second dimension, and that debt, principal, and interest act as input information to the second dimension.

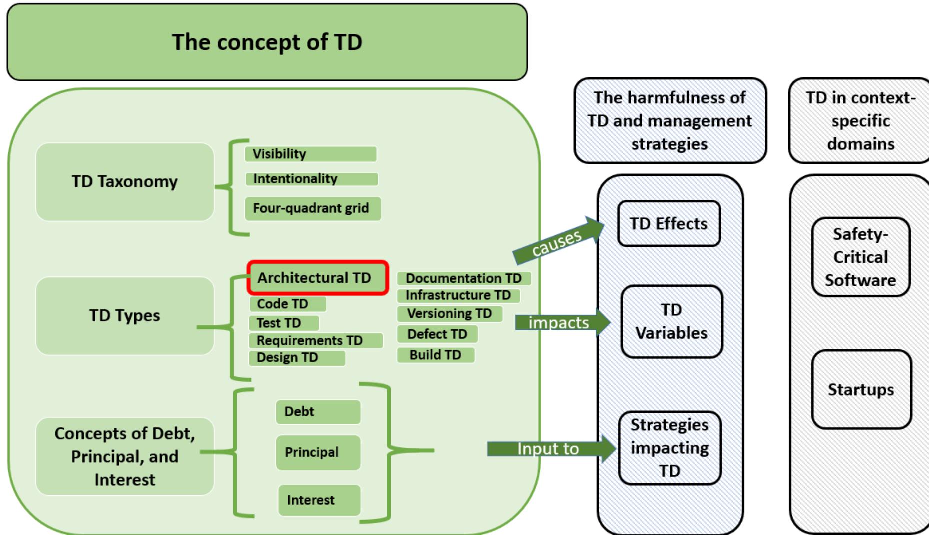


Figure 3: First dimension: The concept of TD

2.1.1. TD taxonomy

As illustrated in figure 3, TD can be categorized in different ways, depending on the perspective adopted. For example, Kruchten et al. [18] provide a categorization based on the visibility of different elements. As illustrated in Figure 4, their model illustrates *visible* elements, such as new functionality to add and defects to fix, and *invisible* elements (those visible only to software developers). Kruchten et al. suggest that only the invisible elements should be considered as TD, where they distinguish between evolution and quality issues.

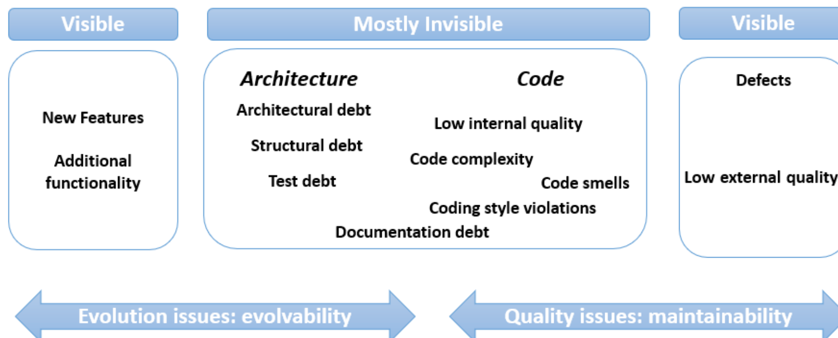


Figure 4: The TD landscape, distinguishing between evolution and quality issues [18].

المنارة للاستشارات
www.r

Yet another classification of the TD landscape is provided by Steve McConnell [19], who categorizes TD based on whether the TD was incurred *intentionally* or *unintentionally*. Unintentionally incurred TD is the non-strategic result of doing a poor job. In some cases, this type of debt can be incurred unknowingly, for example, if a company acquires another company that has accumulated significant TD and which was not identified until after the acquisition. Intentionally incurred TD is commonly found when a company makes a conscious decision to optimize for the present rather than for the future [19].

Similar to McConnell's classification, Martin Fowler [20] provides a categorization, illustrated in Figure 5, where he uses a four-quadrant grid considering the following characteristics: *Reckless, Prudent*, *Deliberate,* and *Inadvertent*. These characteristics comprise what is commonly called the TD Quadrant and allow the classification of the debt by analyzing whether the TD was inserted intentionally or not, and, in both cases, whether it can be considered to be the result of a careless action or was inserted with prudence.

Prudent TD is deliberately introduced because the team is aware of the fact that they are taking on TD and put some thought into whether the payoff for an earlier release is greater than the costs of paying the debt off. A team ignorant of design practices accumulates reckless TD without even realizing the negative consequences of doing so [20].

Martin Fowler argues that *reckless* TD may not be inadvertent. A team could know about good design practices and even be capable of practicing them, but decide to go "quick and dirty" because they think they cannot afford the time required to write clean code.

The last quadrant, *prudent-inadvertent*, refers to the willingness of a team to improve upon whatever has been done, after gaining experience and relevant knowledge.

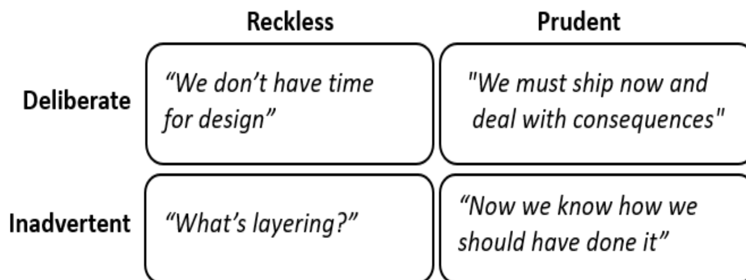


Figure 5: Technical Debt grid quadrant [20].

2.1.2. TD Types

As illustrated in figure 3, there are several different types of TD, and different researchers provide different categorizations for TD types. The TD types presented in the figure are provided by Tom et al. [7]. Similar to such a classification, Li et al. [15] provide an extension of, in total, ten coarse-grained types of TD, including several sub-types: Requirements TD, Architectural TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Versioning TD, and Defect TD. Moreover,

المنارة للاستشارات

www.r

Tamburri et al. [21] map social debt into technical debt, where they link this debt to unforeseen project costs connected to a “suboptimal” development community.

As illustrated in figure 3, TD causes several different negative effects on the software. For instance, in a study by Tom et al. [7], the authors suggest that morale, alongside quality, productivity, are areas that are negatively influenced by the occurrence of TD. Further, as presented in figure 3, different TD types have an impact on different TD variables (e.g., the age of the software or different roles affected by the TD) and Architectural TD (ATD) are highlighted in the figure since this TD type has a specific focus of Papers A and D of this thesis, are commonly described as design decisions that, intentionally or unintentionally, compromise system-wide quality attributes, particularly maintainability and evolvability [22]. More specifically, ATD counts as violations of the code towards the intended architecture for supporting the business goals of the organization [23]. Alves et al. [14] define ATD as referring “to the problems encountered in project architecture, for example, violation of modularity, which can affect architectural requirements (performance, robustness, among others). Normally this type of debt cannot be paid with simple interventions in the code, implying in more extensive development activities.” Similarly, Fernández-Sánchez et al. [24] describe ATD as being caused by shortcuts and shortcomings in design and architecture or by the result of sub-optimal upfront architecture design solutions, that then become sub-optimal as technologies and patterns become superseded.

However, ATD is fraught with several challenges arising from difficulties in detection [25] and the fact that ATD seldom yields observable behaviors to end-users [26], and, even if there are some software tools available for analyzing TD, most of them focus on the code level instead of the architectural aspects of TD [27]. The issue of removing ATD after it has been introduced is often associated with high costs since architectural decisions take many years to evolve and are commonly made early in the software lifecycle, and is often invisible until late in the process [28]. Furthermore, ATD tends to become widespread within the system due to what is known as vicious circles, inferring a non-linear accumulation of the interest with the result of making a later removal even more costly [23]. From a non-technical perspective, ATD is also associated with several challenges, since both managers and other professionals’ awareness of the magnitude of the related consequences of ATD are somewhat limited. This lack of knowledge often leads to the issue that ATD seldom receives sufficient attention from managers and that the allocation of both time and resources to manage and remediate ATD is limited.

2.1.3. Technical Concepts of Debt, Principal, and Interest

The term TD is a financial metaphor, and the most common financial terms that are used in TD research are *debt*, *principal,* and *interest* [29]. These terms are illustrated in Figure 3 together with an arrow that shows that these terms act as input for TD managing strategies in the second dimension.

In financial terms, debt refers to the amount of money owed by one party (debtor or borrower) to another party (creditor or lender) [30], where the obligation of the debtor is to repay a larger sum of money to the creditor at the end of a specified period [31]. The

المنارة للاستشارات

www.r

term debt is used to describe the gap between the existing state of software and some hypothesized “ideal” state in which the system is optimally successful [32].

From an architectural perspective (ATD), this debt refers, for instance, to system shortcomings that can be improved to form an enhanced architectural software quality and to avoid excessive interest payments in the form of decreasing maintainability.

The *interest* refers to the negative effects of the extra effort that have to be paid due to the accumulated amount of debt in the system, such as executing manual processes that could potentially be automated, excessive effort spent on modifying unnecessarily complex code, performance problems due to lower resource usage by inefficient code, and similar costs [7], [25]. Ampatzoglou et al. [30] define interest in their TD financial glossary list as: “The additional effort that is needed to be spent on maintaining the software, because of its decayed design-time quality.”

Financially, the term *principal* refers to the original amount of money borrowed, and, from a software development perspective, the same term is used to describe the cost of remediating planned software system violations concerning TD; in other words, the cost of refactoring [30]. The principal is computed as a combination of the number of violations, the hours to refactor each violation, and the cost of labor [33].

2.2. The impact of Technical Debt in the software development industry

As illustrated in Figure 2, the second dimension of the background model addresses TD from a software development perspective in terms of TD effects, TD variables and strategies impacting TD.

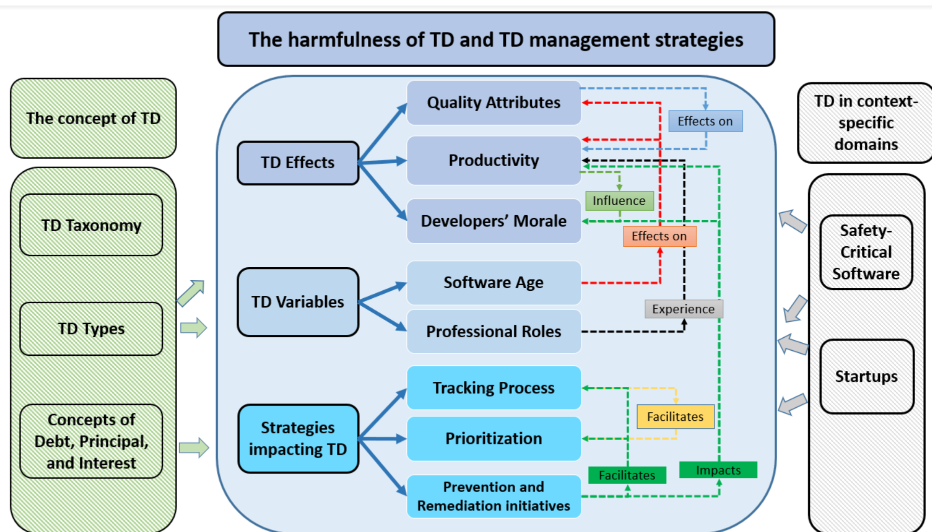


Figure 6: Second dimension: The harmfulness of TD and management strategies

Figure 6 presents a detailed view of the covered aspects of the second dimension, “The harmfulness of TD and management strategies” of the model presented in figure 2. The

المنارة للاستشارات
www.r

dotted lines between the different aspects of this dimension illustrate relationships that are studied in this thesis and further describes in the following sections.

2.2.1. Software Quality Attributes

As illustrated in Figure 6, software suffering from TD negatively affects several different quality attributes, and these affected quality attributes can, consequently, affect the software in different ways. As illustrated by the red dotted line in Figure 6, the level of impact can also vary during the software lifecycle [34],[35], and also cause software developing productivity loss (illustrated by the blue dotted line in Figure 6).

As depicted in Table 1, the software product quality model proposed in ISO/IEC 25010 [36] categorizes product quality properties into eight main characteristics, and each character is composed of a set of related sub-characteristics. This quality model is used in this Ph.D. thesis when accessing how TD negatively affects the overall quality. Li et al.'s [15] systematic mapping study shows that most examined studies argue that TD negatively affects maintainability and that other quality attributes are only mentioned in a handful of studies.

TABLE 1 - SOFTWARE QUALITY ATTRIBUTES - ISO/IEC 25010

Functional suitability Completeness/Correctness/Appropriateness	**Reliability** Maturity/Availability/Fault Tolerance/Recoverability
Performance efficiency Time behavior/Resource Utilization/Capacity	**Security** Confidentiality/Integrity/ Non-repudiation/Accountability/ Authenticity
Usability Appropriateness/Recognizability/ Learnability/Operability/User Error Protection/User Interface Aesthetics/Accessibility	**Maintainability** Modularity/Reusability/ Analyzability/Modifiability/ Testability
Compatibility Co-existence/Interoperability	**Portability** Adaptability/Installability/ Replaceability

2.2.2. Software Age

By definition, software systems are highly evolved products, and there is a commonly held belief that the negative effects of a complex architectural design, in terms of ATD, increase with the age of the software, which is related to the concept of *software aging*

المنارة للاستشارات

www.r

[37]. Parnas [37] argues that software aging is inevitable, yet can be controlled or even reversed. Parnas highlights the causes of software aging, such as obsolescence, incompetent maintenance engineering work, and the effects of residual bugs in long-running systems [38]. “Programs, like people, get old. We cannot prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable.” Furthermore, Mens et al. [39] describe that the negative effects of software aging have a significant economic and social impact on all sectors of industry, and it is therefore crucial to develop tools and techniques to reverse or avoid the intrinsic problems of software aging. This notion is echoed by Lindgren et al. [40], who state that *“*Technical debt refers to software aging costs that are not attended to, which hence need to be repaid at a later time.”
As illustrated with red dotted lines in Figure 6, this thesis includes studies that explore the relationships between the age of the software with both productivity and quality attributes.

2.2.3. Software Professional Roles

Today, there are several different kinds of professional roles present in the software industry, such as e.g., developers, testers, architects, and product managers. These roles have different working tasks and responsibilities and work in different areas and different development phases. The different roles can also have a different background, education, understanding, and scope of knowledge. As illustrated by the black dotted line in Figure 6, several different professional roles participate during the software lifecycle, and their productivity could subsequently be affected by TD. The studies within this Ph.D. thesis involve several different software professional roles that are affected by TD, as well as the roles that are empowered to make decisions in the context of TD.

2.2.4. Tracking Process

Software tooling is a necessary component of any TD management strategy [16], and the tracking process of TD is crucial for the ability to manage TD proactively. Even if there are some tools available (e.g., SonarQube), these tools usually only focus on *identifying* TD at a code level, and these code-focusing tools generally cannot prove indicative for, for example, architectural trade-off, since they can produce misleading results [41]. The available tools also rarely provide the user with any supporting information about the principal or the interest of the TD. Despite the significant need for supporting tools and methods for analyzing TD and ATD, no supporting software tools that iteratively include the measuring, evaluation, and tracking of all the different types of TD exist today. The process of starting to track TD requirements includes both costs in terms of initial investments, and educational and preparation activities. Further, the information collected during the tracking process such as e.g. the debt, the interest, and the principal cost facilitates the TD prioritization process. This relationship is illustrated in Figure 6, by the yellow dotted line.

للاستشارات المنارة
www.r

2.2.5. Software Development Productivity

Several publications, such as [42], [7], [15], state that TD can, in general, have a negative effect on overall software development productivity, yet these publications rarely define what productivity refers to and in what way this reduced productivity can be measured. There is no commonly agreed definition upon the term productivity [43], though in software engineering productivity is commonly defined from a financial perspective, as the effectiveness of productive effort measured as the rate of output per unit of input [43], [44], [45],[30]. Productivity is also a measure of the quality of an output relative to the input required to produce the output. This means that productivity is a combined measurement of efficiency and quality. Software systems suffering from TD cause an extensive amount of *wasted* working time since practitioners are forced to perform additional activities, which would not be necessary if the TD was not present.

In general, there are different ways of measuring software development productivity [46], though in this Ph.D. thesis, we refer to productivity as "the ability to deliver high-quality customer value in the shortest amount of time." This means that the less time that is wasted due to experiencing TD, the greater the increase in productivity, inferring that practitioners can thus use more time focusing on delivering customer value.

The amount of waste of working time due to experiencing TD may vary depending on e.g., different roles and different age of the software. These relationships are presented in Figure 6, where the black dotted line illustrates that this thesis includes studies that explore how different professional roles experience an impact on their productivity due to TD. Similarly, the red dotted lines illustrate that this thesis includes studies that explore how the age of the software affects productivity loss. Further, different compromised software quality attributes may also impact software development productivity, and this relationship is illustrated by the blue dotted line in Figure 6.

2.2.6. Developers' morale

In addition to technical and financial consequences, TD can also affect developers' morale [7], [13]. This association is illustrated in Figure 6 by the green dotted line, which demonstrates that the research presented in this thesis includes studies that explore the relationship between productivity loss due to TD and developer's morale. The reason for this relationship is primarily because the occurrence of TD could hamper the developers from performing their tasks and achieving their developer goals. The term *morale* can be found within the research field of organizational sciences, management, education, and healthcare [47]. Despite the vast body of related literature, the term morale lacks a coherent and precise definition, and Hardy [47] describes how several concepts, such as satisfaction, motivation, and happiness, are commonly used interchangeably to highlight the term morale. In this thesis, we have used the following definition of morale, provided by Hardy [47]: "a cognitive, emotional, and motivational stance toward the goals and tasks of a group. It subsumes confidence, optimism, enthusiasm, and loyalty as well as a sense of common purpose". Furthermore, we adopt an approach for exploring the levels of morale from a set of factors that influence morale, suggested by Hardy [47],

للاستشارات المنارة

www.r

where the antecedent factors of morale are divided into three main categories: affective antecedents, future/goal antecedents, and interpersonal antecedents.

2.2.7. Prioritization of Technical Debt

The decision-making regarding *if* and *when* a TD item should be refactored is part of the TD prioritization process. Several papers propose different prioritization approaches to assist in this process [48], [49], [50], [51], [42], [52].

When TD items are identified, they are commonly registered in some sort of backlog. The used backlogs could, e.g., be a dedicated backlog for only TD items or in a feature backlog where TD items are mixed with features (dependent on different adopted development process). When making decisions on TD prioritizations, cost and value estimations of refactoring initiatives are essential, since these estimations assist in planning the work processes and also prevent potential cost and schedule overruns. As illustrated by the yellow dotted line in Figure 6, this information is commonly derived from a TD tracking process.

However, several authors highlight the difficulties of obtaining such reliable estimates [53], [54], [55], and numerous researchers such as e.g., [56] and [52], state that decisions related to TD are largely based on a manager's gut feeling, rather than hard data gathered through appropriate measurement.

2.2.8. Prevention and Remediation initiatives

There are different prevention and remediation initiatives that can be undertaken to reduce the harmfulness of TD. Such initiatives could e.g., be based on mechanisms where managers impact practitioners' work by adopting different types of incentive programs, and thereby influence software engineers' work outcome, their attitudes, and their work behaviors [57].

As illustrated by the green dotted lines in Figure 6, these prevention and remediation initiatives could have an impact on how practitioners prioritize and track TD and also on the developer morale and the developer productivity.

2.3. Technical Debt in context-specific domains

As illustrated in Figure 2, the third dimension of the background model focuses on TD in two different context-specific domains: the startup company domain and the safety-critical domain.

المنارة للاستشارات

www.r

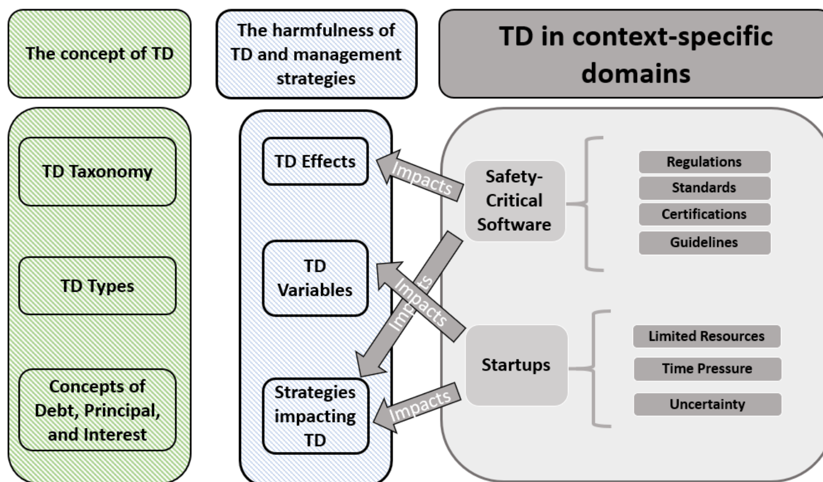


Figure 7: Third dimension: TD in context-specific domains

Figure 7 presents a detailed view of the covered aspects of the third dimension “TD in context-specific domains” of the model presented in figure 2. This figure illustrates that this thesis includes research addressing the impact of Safety-Critical Software aspects on both TD effects and TD managing strategies. The figure also illustrates that the thesis explores how Startups strategically manage TD and its relationship to TD variables.

2.3.1. Safety-Critical Software

Different types of software systems operating in different domains have different types of guidelines and regulatory requirements to which the software must adhere before it can be placed on the market. Software systems operating in the safety-critical software (SCS) domain are, for instance, heavily regulated and require certification against industry standards.

As illustrated in Figure 7, these SCS standards have an impact on different TD effects and TD management strategies, and for instance Ghanbari [58] states that these standards can have a significant impact on the process of conducting refactoring tasks. Further, the architecture may also have a high impact on the characteristics of the system under its development [59]. However, these safety-critical regulations may also strengthen the implementation of both source code and architecture, and thereby initially limit the introduction of TD.

2.3.2. Startups and Technical Debt

Giardino et al. [6] define software startups as those “organizations focused on the creation of high-tech and innovative products, with little or no operating history, aiming to aggressively grow their business in highly scalable markets.” Software startups are newly created companies, typically with no operating history, and are mainly oriented towards developing high-tech and innovative products, aiming to grow their business in highly

المنارة للاستشارات

www.r

scalable markets [60], [6]. Compared to more mature companies who often maintain the software in an established market, software startups face different types of challenges. As illustrated in Figure 7, the software development approach in Startup contexts has an impact on TD management strategies. This impact can be described in terms of that Startups often operate with limited resources and under extreme time pressure as they strive to produce their product and avoid being beaten to market by a competitor or running out of capital [3]. This pressure is likely to cause startups to accumulate TD as they make decisions that are focused more on the short-term than the long-term health of the codebase, which requires later attention and need to be managed carefully. Further, the context which startups operate within can also have an impact on different TD variables such different roles and different phases during the startup lifecycle.

المنارة للاستشارات

www.r

المنارة للاستشارات

www.m

3. Research Motivation

As highlighted in the previous Introduction chapter, software systems and software development processes suffering from TD can be impeded in terms of the technical, financial, and developer working situational perspectives. However, since limited knowledge and few supporting tools are available to measure the extent of TD within a system, it is quite difficult to compute the negative effects that TD causes in terms of, for example, extra costs, extra activities, and the need for extra resources. Without this knowledge, software development organizations are not aware of the interest that they are paying on the debt, and therefore they might not currently give TD management the necessary attention within their organizations. Furthermore, without this information, software organizations risk not focusing sufficiently on prevention mechanisms and deliberate remediation of their TD, which, over time, can result in high defect rates, project delays, quality complications, and very low developer productivity.

Although significant theoretical work has been undertaken to describe the negative effects of TD, to date very few *empirical* studies focus on these effects' impact and consequences for software development. Therefore, there is a need for more *empirical* assessments in the research field, with a focus on quantifying the negative effects and on providing a more in-depth understanding of its related negative consequences.

The overarching goal of this Ph.D. thesis is threefold. The first goal is to *empirically* study and understand *in what way* and *to what extent* TD influences today's software development work, specifically with the intention of providing more *quantitative* insights into the field. Second, the goal is to understand which different initiatives can reduce the negative effects of TD and also which factors are important to consider when implementing such initiatives in the industry. The third goal is to study TD in different context-specific domains in order to understand if companies developing different types of software manage and perceive TD differently.

More specifically, for the first goal, we explore the negative effects of TD from four different perspectives.

Additionally, for the second goal, and based on the findings from the first, we synthesize different initiatives that can be undertaken in order to reduce the negative effects of TD.

The third goal is addressed by studying TD in two different domains, a software startup company domain, and a safety-critical software domain. In order to address this goal, this thesis studies two different context-specific domains; the software startup domain and the safety-critical software domain. The rationale for selecting these specific domains is related to the commonly widely different ways in which these types of companies work with software development. Where, for instance, developing software for safety-critical applications are commonly heavily regulated and require certification against industry standards. Meanwhile, the software development process in startup companies quite often is less strict compared to more mature software development companies and has, in general, focus on the speed of the development in order to get the software out on the market as quickly as possible.

المنارة للاستشارات

www.r

The different studied perspectives, initiatives, and domains are presented in Figure 8.

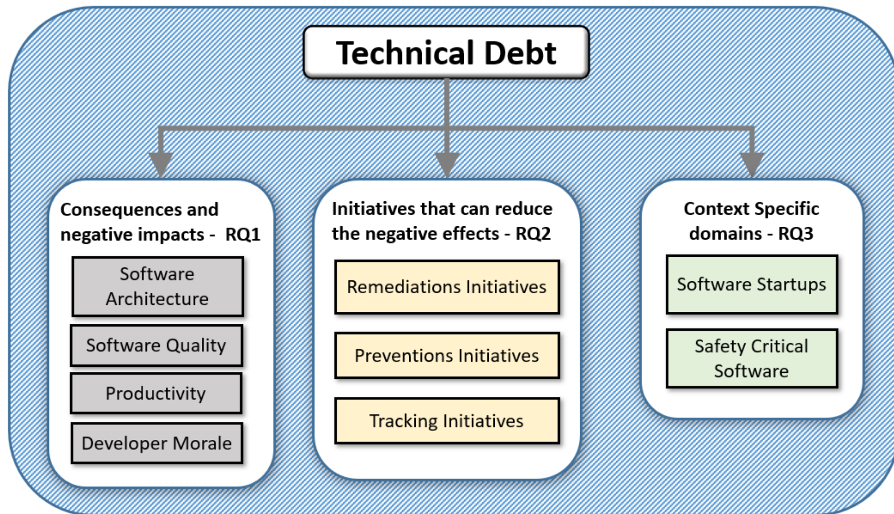


Figure 8: The overarching research goal in this thesis.

المنارة للاستشارات

www.r

4. Research Questions

Based on the research motivation presented in Chapter 3, three main research questions are derived, together with a total of nine sub-questions. These research questions are the primary drivers of the research studies presented in this Ph.D. thesis. The research questions (RQ) are presented in Table II.

TABLE II - RESEARCH QUESTIONS

Main Research Question (RQ)	Sub-question	Research Question
1	**What are the consequences of TD in today's software industry?**	
	1.1	What is the negative impact of *architectural* TD?
	1.2	What is the negative impact on *software quality* due to TD?
	1.3	What is the negative impact on development *productivity* due to TD?
	1.4	What is the negative impact on developers' *morale* due to TD?
2	**What initiatives reduce the negative effects of TD in today's software industry?**	
	2.1	What impact do *TD prevention* initiatives have on software development?
	2.2	What impact do *TD remediation* initiatives have on software development?
	2.3	What impact do *TD tracking* initiatives have on software development?
3	**What factors affect TD management in context-specific domains?**	
	3.1	What are the challenges and benefits of *deliberately introducing TD* for software startups?
	3.2	What impact can *regulatory requirements* have on TD management in a safety-critical software context?

المنارة للاستشارات

www.r

5. Relationship between Studies and Research Questions

This Ph.D. thesis presents eleven research studies (Papers A–K), where each individual research study fully or partly addresses the research question described in Chapter 4. Figure 9 provides an overview of the relationships between the studies and the research questions. The figure also illustrates in which chapter the papers are presented in. The findings for each research question are described in Chapter 20.

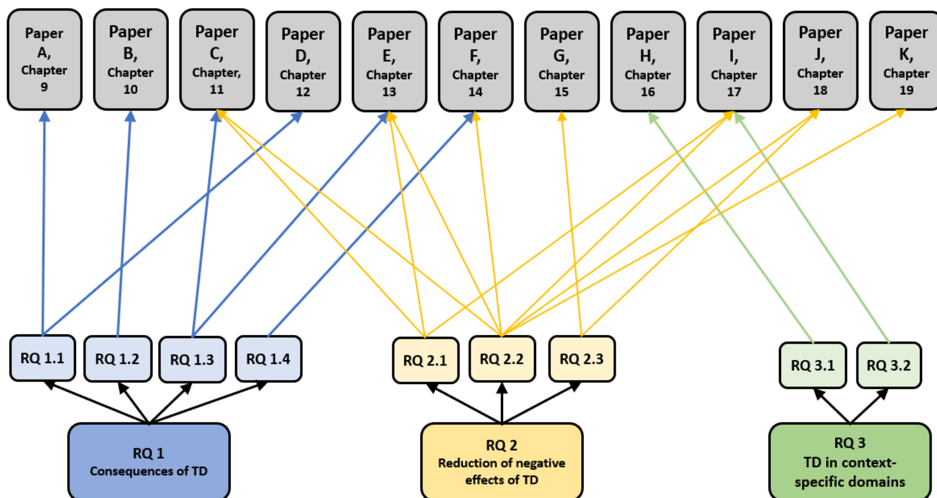


Figure 9: Research studies and findings presented in the thesis.

5.1. Research studies addressing RQ1

As presented in Figure 8, the first main research question (RQ1) sets out to understand the consequences of TD in today's software industry, as seen from different perspectives. This research question was addressed in six studies:

- *Managing architectural technical debt: A unified model and systematic literature review*
- *Time to Pay Up – Technical Debt from a Software Quality Perspective*
- *The Pricey Bill of Technical Debt – When and by Whom Will it be Paid*
- *Impact of Architectural Technical Debt on Daily Software Development Work – A Survey of Software Practitioners*
- *Software developer productivity loss due to technical debt – A replication and extension study examining developers' development work*
- *The Influence of Technical Debt on Software Developer Morale*

المنارة للاستشارات

www.r

5.2. Research studies addressing RQ2

As presented in Figure 8, the second research question (RQ2) explores which different initiatives reduce the negative effects of TD in today's software industry. This research question was addressed in seven studies:

- *The Pricey Bill of Technical Debt – When and by Whom Will it be Paid?*
- *Software developer productivity loss due to technical debt – A replication and extension study examining developers' development work*
- *The Influence of Technical Debt on Software Developer Morale*
- *Technical Debt Tracking: Current State of Practice – A Survey and Multiple Case-Study in 15 Large Organizations*
- *How Regulations of Safety-Critical Software Affect Technical Debt*
- *Technical debt triage in backlog management*
- *Carrot and stick approaches when managing Technical Debt*

5.3. Research studies addressing RQ3

As presented in Figure 8, the third research question (RQ3) examines different factors that affect TD management in context-specific domains. This research question was addressed in two studies:

- *Embracing Technical Debt, from a Startup Company Perspective*
- *How Regulations of Safety-Critical Software Affect Technical Debt*

المنارة للاستشارات
www.m

المنارة للاستشارات

www.m

6. Methodology

Software engineering is a multi-disciplinary field, encompassing not only technological but also social boundaries. Therefore, not only do the tools and processes software engineers use need to be investigated, but also the social and cognitive processes surrounding them, which includes the study of concerned professionals, their working tasks, and activities. Thus, we need to understand how individual software engineers develop software, as well as how teams and organizations coordinate their efforts [61].

This thesis includes eleven publications, and, in order to fulfill the goals of this Ph.D. thesis, different research methods and different research approaches have been adopted. Table III provides an overview of the goals together with the selected research types, the research approaches, and, finally, the research methods used for each of the included publications. It is apparent from this table that this Ph.D. thesis has a strong emphasis on empirical research, where most of the analyzed data are based on estimated and/or reported artifacts and derive knowledge from actual industrial settings and experiences, rather than from theories or anecdotal evidence. It can also be seen in Table III that a strong focus is placed on combining both a qualitative and quantitative research methodology using a mixed-methods approach.

TABLE III - OVERVIEW OF THE INCLUDED PUBLICATIONS

Paper	Goal/Focus area	Research Type	Research Approach	Research Method
A	ATD Composition*	Systematic Literature Review	Qualitative (and Quantitative)	Systematic Literature Review (SLR)
B	Affected Quality Attributes due to TD	Empirical Approach	Mixed Methods	Interviews (n = 43+32) and Survey (n = 258)
C	Quantification of the *Estimated* Interest of TD	Empirical Approach	Mixed Methods	Interviews (n = 32) and Survey (n = 258)
D	Software Practitioners' Perception of the Impact of ATD*	Empirical Approach	Quantitative	Survey (n = 258)
E	Software Developer Productivity Loss due to TD Including a Replication Study of *Reported* Data	Empirical Approach	Mixed Methods (longitudinal + replication study)	Interviews (n = 16) and Survey (n = 43; 473 datapoints) + Survey (n = 47; 177 datapoints)

المنارة للاستشارات

www.r

F	The Influence of TD on Software Developer Morale	Empirical Approach	Mixed Methods	Interviews (n = 15) Survey (n = 33) Survey (n = 473)
G	Introduction of Tracking TD	Empirical Approach	Mixed Methods	Interviews (n = 13) and Survey (n = 226) Document analysis
H	TD from a Startup Perspective	Empirical Approach	Quantitative	Interviews (n = 16)
I	Safety-Critical Software and TD	Empirical Approach	Quantitative	Interviews (n = 19)
J	TD Prioritization in Backlogs	Empirical Approach	Mixed Methods	Interviews (n = 17) and Survey (n = 17)
K	TD Management Strategies	Empirical Approach	Mixed Methods	Interviews (n = 32) and Survey (n = 258)

*This study has a specific focus on ATD

6.1. Research Approaches

The included studies that form this thesis use different research approaches. The approaches adopted are listed in this section, together with a short description as well as the benefits of each approach.

6.1.1. Qualitative research

The goal of conducting qualitative research is the "Development of concepts which help us to understand social phenomena in natural (rather than experimental) settings, given due emphasis to the meanings, experiences, and views of the participants" [62]. The motivation for using this qualitative research approach was to obtain richer information, to gain more in-depth insights into the studied phenomenon, and to understand the perceptions that underlie and influence different studied negative effects. In this thesis, we have chosen individual interviews, group interviews, and documents as the data collection approaches when conducting qualitative research.

6.1.2. Quantitative research

The goal of conducting quantitative research is to "explain behavior in terms of specific causes (independent variables) and the measurement of the effects of those causes (dependent variables)" [63]. The benefits of a quantitative research approach include improving the generalizations of a larger number of subjects and to thereby achieve higher objectivity. The quantitative data collection method used in this thesis is based on surveys.

المنارة للاستشارات

www.r

6.1.3. Mixed-Methods research

A mixed-methods research approach involves the collection of both qualitative and quantitative data, where the two forms of data collection are integrated into the design through merging the data, connecting the data, or embedding the data. The purpose of this approach is to provide a complete understanding of the phenomena being studied [64]. It can be argued that the benefits of a mixed-method approach includes the provision of a stronger understanding of the problem than either method on its own, and by minimizing the limitations of both approaches [65]. An advantageous characteristic of conducting mixed methods research is the ability to perform triangulation. However, there is a potential weakness of mixing methods for the purpose of validity convergence, namely to compare outcomes from different methods to see if they agree, as the interpretation of agreement or disagreement is not straightforward [64].

The mixed-methods research approach used in this thesis has contributed to a comparison of different perspectives drawn from both qualitative and quantitative data within the same studies. This approach has also assisted in explaining quantitative results with qualitative follow-up data collections. Even if it is claimed that it is more difficult to execute studies based on a mixed-methods approach [66], the motivation for using this approach was to be able to address more complex research questions and to collect a richer and stronger array of evidence than could be accomplished by using a single method alone [66].

When interpreting the results from a mixed-methods research approach, there are different designs to facilitate the provision of a stronger interpretation and obtain more insight from the results. This thesis has used different typologies for the classification of different mixed-methods strategies. The *convergent parallel mixed-methods* design was used in Paper C, where we collected both qualitative and quantitative data, analyzed them separately, and compared the results to understand whether the findings confirmed or contradicted each other. In Papers E, F, G, and K, an *explanatory sequential mixed-methods design* was used, where, as a first step, we collected and analyzed the quantitative data and used this result on which to build the qualitative data collection. In Papers B, I and J, we first collected and analyzed the qualitative data and, after that, collected the quantitative data, using a so-called *explanatory sequential mixed method design.*

6.1.4. Longitudinal studies

A longitudinal study is a research method that contains repetitive observations of the same variables (e.g., time usage) on more than one occasion and over time [67]. The incentive for using a longitudinal research method in this study (Paper E) has two principal aspects: a) To increase the precision of reporting experienced data (in our case, not based on single estimations and single perceptions). This was achieved by studying each respondent for several weeks, where the reported data could be compared. Such designs are called repeated-measures designs [67]; and b) To examine the respondents' changing responses over time. Longitudinal designs have a natural appeal for the study of changes associated with development or changes over time. They have value for describing both temporal changes and their dependence on individual characteristics [67]. Ployhart and

المنارة للاستشارات

www.r

Vandenberg [67] state that "Longitudinal designs give greater precision per observation, but observations may be more expensive or difficult to collect. Problems with missing or suspect data may be harder to solve in longitudinal studies. Implementation issues also influence design, since it is not always possible to sustain the commitment of investigators and participants or the quality of study procedures".

To address the potential problem with missing data from the respondents, for instance, if the respondents for some reason did not enter the data in one or more surveys, they were always asked to report their experienced data since the last time they took the survey. This wording means that if for some reason, the respondent did not enter the data in one or more surveys, they would enter the data from the last time the respondent took the survey. In this way, the surveys cover the full period of sampling.

6.1.5. Replication studies

Replication plays a key role in empirical software engineering [68] and is proposed as an important means of increasing confidence in and assessing the reliability of results [69], [70]. A common goal of conducting a replication of a study is to assist a research community in gaining information about conditions under which the original set of results hold [71], [72].

Paper E includes an additional replication study with the goal of investigating whether the original results demonstrate that the findings can be generated repeatedly, and that the original findings were thus not an exceptional case. This replication study was characterized as an "Internal Exact Independent Partly" study. *Internal* reflects that the replication team was the same as in the previous phases. The categorization *Exact* reflects that the procedures were followed as closely as possible to determine whether the same results could be obtained. The categorization of *Independence* indicates that, during the replication phase of the study, we deliberately varied some aspects of the conditions when collecting the data [68]. Finally, the categorization *Partly* points out that not all of the research areas and research questions were replicated.

6.1.6. Data Collection

The collected data in this Ph.D. thesis consist of both primary and secondary studies, where the primary form of data collection is original research, and where the secondary study sets out to aggregate and synthesize the outcomes of other primary studies in an objective and unbiased manner using either a qualitative or quantitative form of synthesis. The secondary study in this thesis refers to the systematic literature review presented in Paper A. The adopted quantitative methods involve surveys, and the qualitative methods contain interviews and, to some extent, as well as the analysis of documents and reports from various tools, which are presented in Papers B–K (see Table III).

6.1.7. Interviews

The data collection method in this thesis includes several interviews with industrial practitioners within the software engineering field, where we, as researchers, asked a

المنارة للاستشارات

www.r

series of questions to a set of subjects regarding the areas of interest in the study. This thesis includes both interviews with a single interviewee well as several group interviews (focus groups), with several interview subjects simultaneously. According to the guidance provided by Runeson and Höst [73], during all interviews, the dialog between the researcher and the subject(s) was conducted by a set of pre-defined interview questions.

Runeson and Höst [73] distinguish between unstructured, semi-structured, and fully structured interviews. Unstructured interviews are a very flexible approach whereby the area of interest is established by the interviewer, but the discussion of the issues is guided by the interviewees [74]. In fully structured interviews, the interviewer has full control of the order of the questions, which are all predetermined [74]. A fully structured interview is similar to a face-to-face completion of a survey [73]. The interviews conducted in this Ph.D. thesis are all semi-structured in nature, with the advantage of allowing for improvisation and the exploration of the studied objects [73].

Semi-structured interviews include a combination of open-ended and closed questions, designed to elicit not only the anticipated information, but also other information not foreseen by the interviewer. In semi-structured interviews, questions are planned but are not necessarily asked in the same order as they are listed in the interview protocol [73]. We used semi-structured interviews with the intention of ensuring that they provide us with valuable results, since the interviewees' awareness and knowledge about the concept of TD could potentially differ considerably, and therefore it was important to carefully explain the concepts used in order to create a comparable understanding between the interviewer and the interviewees.

For accuracy, all interviews were digitally recorded and transcribed verbatim (all interviewees were asked for recording permission before starting). All interviews were treated anonymously, regarding both the name of the interviewee and the company name.

All interviews were based on a selective sampling of the interviewees, with respect to their role and their expertise. Several of the publications included in this thesis include interviews with individuals in software roles, such as software architects, developers, testers, project managers, and product managers.

Some of the conducted interviews were characterized as "Follow-Up" interviews, meaning that, to some extent, they focused on corroborating certain findings that we already thought had been established during previous data collection activities, where the questions were carefully worded (avoiding leading questions) to allow the interviewee to provide fresh commentary on, for example, previously presented material [66].

The study in Paper E also includes a pre-study. During this initial pre-study, the motivation for the study was presented and discussed with software practitioners from seven software companies within our network, including an extensive range of software development. This phase acted as a guide for collecting information concerning the studied context and for selecting the most appropriate research model.

The interviews in Papers C, G, E, F, and K were conducted with interviewees who had previously answered one or more surveys, and during these interviews, the compiled results from the interviewees' individual results from the survey were presented. During interviews with their managers as well as during group interviews, an aggregated view of

المنارة للاستشارات
www.m

all the respondents from the respective company was presented. This presentation allowed the interviewees to relate to the interview questions more easily, where the results of the survey were addressed. The interview questions for these studies were designed to a) increase the understanding of the survey results, b) ensure that the questions in the survey were understood and interpreted as intended and also uniformly, c) confirm the results from the survey, and d) understand the implications of the survey results.

6.1.8. Surveys

Initially, we would like to clarify the term "survey" in this Ph.D. thesis. A survey, in this context, refers to the questionnaire (to differentiate it from a "survey" as a literature review). Surveys are considered one of the most common data collection methods in software engineering research. Surveys aim to achieve generalizability over a certain population, for instance, different software developing practitioners or end-users [75]. Their advantages can be described in terms of facilitating the recruitment of respondents, who can be anonymous, since anonymity is believed to help gain access to respondents who would normally be hard to reach, and it may facilitate the sharing of their experiences and opinions. Online surveys are considered useful when the issues being researched are particularly sensitive [76].

The motivation for using surveys in this thesis was to reach a high level of generalizability regarding a large number of software professionals, and to maximize coverage and participation without having to conduct time-consuming interviews. We also aimed to collect data for quantitative analysis that could contribute to a more detailed examination of the different relationships and aspects of the studied topics. Aside from Papers A, H, and I, all papers included in this thesis incorporated a research method that, to some extent, included a survey. According to the guidance provided by [77], the drafts of all surveys were first tested by at least one industrial practitioner and by one Ph.D. candidate in order to evaluate the understanding of the questions and the usage of common terms and expressions. During this evaluation, we also monitored the time needed to complete each survey. All surveys were designed and hosted by the online survey service SurveyMonkey.

Except for the surveys used in the longitudinal study in Paper E, all the surveys used a mix of open-ended and closed questions where the respondents could either select an answer from among pre-defined alternatives or formulate their answers freely in a text field. The questions were a combination of optional and mandatory. To avoid bias in these surveys, the questions were developed as neutrally as possible, ordered in such a way that one question did not influence the response to the next question, and a clarifying description was provided when needed [78].

6.1.9. Systematic Literature Reviews

A vital part of conducting software engineering research is the ability to identify existing research on technologies, tools, theories, and methods in order to evaluate and make informed and scientific decisions. Empirical approaches that include systematic review methodologies such as a systematic literature review (SLR) are effective in this context

المنارة للاستشارات

www.r

[79]. The main rationale for undertaking an SLR is to synthesize existing work and to identify gaps in current research in order to suggest areas for further research [80].

The SLR process should be carried out in accordance with a predefined search strategy, which allows the search to be assessed. The major advantage of using this method is that the result is provided by evidence which is robust and transferable, and that sources of variation can be further studied [80]. It provides a framework for establishing the importance of the study as well as a benchmark for comparing the results with other findings [65].

المنارة للاستشارات

www.r

المنارة للاستشارات

www.m

7. Data analysis

The data analysis in this thesis has been carried out in different ways, depending on the type of data that was collected. Quantitative data are analyzed using statistics, while the qualitative data are analyzed using categorizations and sortings [73].

7.1. Quantitative Data Analysis

Techniques for analyzing and summarizing quantitative data include different methods, such as determining measures of central tendency (e.g., median and mean) and measures of dispersion (e.g., ranges and standard deviations) [81].

For example, the collected data from the surveys used in this thesis were analyzed in a quantitative fashion, i.e., by interpreting the values obtained from the answers. All analyses were carried out using the software package SPSS (version 22), R (version 3.3.2) [30], and by using the optional collection of R packages from tidyverse (version 1.1.1) [32] for data manipulation and visualization. The data collected in the surveys were, e.g., analyzed by assessing the median, mean and standard deviation and also by using statistical methods such as Pearson's R correlation coefficient, the Pearson chi-square tests, F-tests, Holm's procedure, the Wilcoxon signed-rank test, and ANOVA.

Pearson's R method was used for correlation analysis of associations between variables. This method computes pairwise, determining the strength and direction of the association between two values, and can be used to describe a linear relationship between two values. Pearson chi-square tests were used for evaluating the likelihood of any observed difference between the values arising by chance, and for assessing whether unpaired observations of two variables were independent of each other. F-tests using Satterthwaite's approximation of the denominator degrees of freedom were used for significance tests of regression coefficients [82]. Holm's statistical procedure [83] was used to counteract the problem of multiple comparisons of different groups of data. The Wilcoxon signed-rank test is a non-parametric statistical hypothesis test and was used when comparing two related samples when the population could not be assumed to be normally distributed. The one-way ANOVA test was used to compare the means of two or more independent groups in order to determine whether there was statistical evidence that the associated means were significantly different.

7.2. Qualitative Data Analysis

When analyzing the qualitative data collected in this thesis, a thematic analysis approach was used. Thematic analysis is a method for identifying, analyzing, and reporting patterns and themes within data, which involves searching across a dataset to find repeated patterns of meaning. The thematic analysis provides a flexible and useful research tool, which offers a detailed and complex account of the collected data [84].

When analyzing the qualitative data, guidelines provided by Braun and Clarke [84] were used in order to conduct the analysis in a thorough and rigorous manner. The thematic analysis was conducted using a six-phase guide. First, the audio recorded qualitative data

المنارة للاستشارات

www.m

collected from interviews were transcribed into written form, where we were also able to familiarize ourselves with the data. The second step involved the production of initial codes from the data, where we organized the data into meaningful groups. In this phase of the analysis, a Qualitative Data Analysis (QDA) software package called Atlas.ti was used. The third phase focused on searching for themes by sorting the different codes into potential themes and collating all the relevant coded data extracts within each identified theme. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the citation "Maybe you have to encourage the developers a bit, to get the data" was coded as "Willingness to input data" in the theme of "Measuring Wasted time Aspects" in Paper E. To ensure that the coding was performed in a consistent and reliable fashion, and in order to triangulate the interpretation of the data and to avoid bias as much as possible, at least two authors synchronized the output of the coding in the papers, following guidelines provided by Campbell et al. [85].

The fourth phase focused on the revised set of candidate themes, involving the refinement of those themes. The refinement focused on forming coherent patterns within the themes. Otherwise, we revised the themes or created a new theme. The fifth phase focused on identifying the essence of each theme and determining what aspect of the data is captured by each theme. This phase also stressed the importance of not just paraphrasing the content of the data extracts, but also identifying what is interesting about them and why. The final phase of the thematic analysis took place when we had a set of fully developed themes, and involved the final analysis and write-up for publication.

7.3. Threats to Validity

The validity of a research study refers to the trustworthiness, credibility, conformability, and data dependability of the results, and to what extent the results are reliable and not biased by the researchers' personal opinions [73], [66]. In this thesis, we have chosen a classification scheme in order to distinguish between different aspects of validity and threats to validity provided by Runesson and Höst [73], which is also used by Yin [66] and Wohlin et al. [86]. This scheme distinguishes between four aspects of validity: *construct* validity, *internal* validity, *external* validity, and *reliability.*

7.3.1. Construct validity

Construct validity addresses the extent to which operational measures that are studied represent what the researchers are considering as well as the desire to investigate according to the research questions [73], i.e., whether the theoretical constructs are interpreted and measured correctly [61]. The findings presented in this Ph.D. thesis could potentially be affected by some threats to construct validity, and in order to mitigate this risk, several different approaches were employed, depending on the type of study.

For example, in the longitudinal study in Paper E, this risk was mitigated by trying to ensure that all the participants had the same base of knowledge in the field of study; all participants received educational material as a first activity when they joined the study. However, we cannot ensure that all the participants read and understood the material.

المنارة للاستشارات

www.r

Another example is the studies that used web surveys as part of the data collection. In these studies, the construct validity threat was addressed by helping the respondents to distinguish between different types of TD, and a short description of each type of TD was used in the surveys. An additional threat to this validity can stem from the fact that the qualitative data derived from the survey (excluding Paper E), are based on perceptions and estimations (not on measured, reported, or observed data) made by the respondents. Moreover, in Paper A, which is a systematic literature review, we have attempted to mitigate this risk by only including peer-reviewed publications from journals, conference proceedings, or workshop proceedings, and only two peer-reviewed book chapters were included in order to include all relevant publications concerning ATD. In addition, in order to mitigate the risk of not retrieving relevant publications, which could affect the completeness of the study, we searched the most common electronic databases and also conducted both a forward and backward snowballing technique [87].

7.3.2. Internal validity

Internal validity refers to whether the result is correctly derived from the researcher's conclusions without external factors potentially affecting the result. "When the researcher is investigating whether one factor affects an investigated factor, there is a risk that the investigated factor is also affected by a third factor" [73]. Yin [66] states that internal validity primarily concerns studies where the researcher is trying to explain *how* and *why* one event led to another and thereby concludes a causal relationship without considering the presence of additional excluded events. Furthermore, Easterbrook et al. [61] state that it is a common mistake to confuse correlation with causality and that it is much harder to demonstrate causality than to show that two variables are correlated.

The results of this thesis could potentially be affected by this threat since, in some of the included studies, the findings are correlational and also indicate a causal relationship. For example, in the studies where we examine the amount of wasted time, activities, and different TD types, this threat affects our ability to accurately explain the phenomena that we observed [66]. This threat is, for example, demonstrated in Paper B, where we use the estimated wasted time correlated with the frequency of how often the respondent encountered each of the listed quality attributes. By performing this correlation, the validity could potentially have been violated by either finding relationships that are non-existent or missing real relationships that are wrongly deemed non-significant. However, by combining correlation analysis with analytical causality (from, e.g., the conducted interviews) in Papers E, F, G, and K, causality links can be suggested. Therefore, to mitigate this threat, we have in these papers triangulated the data by conducting follow-up interviews validating the derived results.

7.3.3. External validity

The external aspect of validity addresses the extent to which it is possible to generalize the findings of the study and for them to be applicable to other situations [73], where Yin [66] describes generalization as follows: "An analytic generalization consists of a carefully posed theoretical statement, theory, or theoretical proposition." Easterbrook et

المنارة للاستشارات

www.r

al. [61] state that this commonly depends on the nature of the sampling used in a study. This notion is echoed by Morse et al. [88]: "the sample must be appropriate, consisting of participants who best represent or have knowledge of the research topic."

Following Yin's [64] guidelines, it is important to ensure external validity in terms of analytic generalizability. This was taken into account when formulating the research questions by using the terms "*how*" and *"what,"* which increases the possibility of arriving at an analytical generalization. Moreover, in order to mitigate this risk, the goal is to enable analytical generalization where the results are extended to situations that have common characteristics, and thus for which the findings of the study are applicable [73]. Although we cannot generalize the results in this study to all different software companies, we can rely on a high number of participating organizations from different types of software development settings, e.g., different business domains, programming languages, and experience, and also on a relatively high number of participants from each company, with different professional roles. Furthermore, in studies where surveys are used as part of the data collection, there is always a risk that the sample is biased, and a potential threat, therefore, refers to the demographic distribution of response samples. The companies in this thesis are primarily from the Scandinavian region. Without replicating this study in other countries or regions, it is not possible to confirm the generalizability of these results.

7.3.4. Reliability

The goal of reliability in research is to minimize the errors and biases in a study. Reliability addresses whether the study would yield the same results if other researchers replicated them, following the same procedure, by means of the extent to which the analysis is dependent on specific researchers [73], [66]. An example of a threat to the reliability could, e.g., occur if researchers introduce bias into the study where a tool being evaluated is one in which the researchers themselves have a stake [61]. Morse et al. [88] state that it is important to ensure the attainment of rigor and reliability verification strategies early, and also during the whole study, in a proactive manner and not only apply them using a post-hoc evaluation after the research is completed.

An example of a threat to the reliability can be found in the included papers where the results are based on estimated values from participants, while we do not know what the given estimations are based on. However, as the demographic data show in these studies, many participants have several years of software development experience. This means that they are used to estimate the amount of work that has been undertaken or is upcoming, which mitigates the threat that the estimated effort would be very distant from the "actual" one.

During the design of the studies included in this Ph.D. thesis, we attempted to mitigate this threat in various ways. First, one prerequisite to allowing for repeatability is, for example, access to the used surveys, which we have addressed by making all surveys available online. To assist in replicability, we have also reported how the data analysis protocol was constructed for interviews and the SLR. Second, the code and sub-codes of the collected data from interviews are reported. Third, as suggested by [65], the findings

المنارة للاستشارات

www.r

in several papers are presented using a rich and thick description box that "may transport the reader to the setting and give the discussion an element of shared experiences." Fourth, the studies included in this Ph.D. thesis also present negative or discrepant information that runs counter to the studied themes. When presenting contradicting evidence, the account becomes more realistic and more valid, according to [65]. Fifth, in several of the included studies in this Ph.D. thesis, the methodology section initially presents the used research design where each step or phase of the study is both visually and textually described in detail. This information may contribute to the replication of the studies. Sixth, since replication plays a key role in empirical software engineering [68] and is proposed as an important means of increasing confidence in and assessing the reliability of results [69], [70], the findings in Paper E involve a replicational phase. Seventh, we have employed source triangulation, methodological triangulation, and, finally, observer triangulation. The triangulation is presented as a separate section (section 7.5.3, below) in this Ph.D. thesis.

7.3.5. Triangulation

To achieve a higher degree of validity and reliability, we have adopted different triangulation techniques. Triangulation is important in order to increase the precision of empirical research, in terms of adopting different perspectives towards the studied object and thus providing a broader view [73]. In this thesis, three different types of triangulation techniques are mainly used: *source* triangulation, *observer* triangulation, and *methodological* triangulation.

Source (data) triangulation refers to using several sampling strategies to ensure that data is gathered at different times and in different situations. The use of more than one source (e.g., interviews, surveys, documents) of data strengthens the conclusion since it can be drawn from several sources of information [73], [89]. During source triangulation, the sources of evidence can be of either a convergent or non-convergent character, where the *converging lines of inquiry* refer to when more than one source of data corroborates the same finding, while non-convergence refers to when different sources of data address different findings [66]. As illustrated in Table III, this type of triangulation is used in Papers B, C, E, F, G, I, and K, where we have used more than one source of information, such as interviews, surveys, and analysis of documents. In these papers, both convergent and non-convergent source triangulation strategies are used.

Observer (researcher) triangulation refers to using more than one observer to gather and interpret data [73], [89]. This type of triangulation was achieved in all papers included in this thesis, where at least two of the involved researchers worked together in different roles during the studies, thus enabling peer debriefing and the analysis of the collected data. For example, in the SLR in Paper A, two researchers independently examined several of the retrieved publications to ensure that they were suitable and equivalently analyzed. To reduce the risk of subjectivity during the classification and extraction phase, performed by only one researcher, several publications were examined by at least two researchers in order to ensure that the returned publications were suitable and equivalently analyzed.

المنارة للاستشارات

www.r

Methodological triangulation refers to combining different types of data collection methods, such as a combination of both qualitative and quantitative methods [73], [89]. According to [65], the goal of using different types of data collection is that this triangulation strategy “has the potential of exposing unique differences or meaningful information that may have remained undiscovered with the use of only one approach or data collection technique in the study”. As illustrated in Table III, this type of triangulation was used in Papers B, C, E, F, G, I, and K, where we have used more than one type of data collection method.

المنارة للاستشارات
www.r

8. Overview of Papers and Findings

In order to provide an overview of the work presented in this thesis, this chapter presents the studies included in this Ph.D. thesis and explains how they are related.

Each paper is described in two sub-sections, where the first section (study summary) describes the study's goal, the research questions addressed, the selected research approach, the research category, and, finally, the research method used. In the second sub-section (results), the results and contributions from each of the included studies are presented and synthesized. This second sub-section also sets out to discuss how the papers are related and describes how each of the papers contribute to answering the research question formulated in chapter 4 and presented in Figure 9.

8.1. Paper A: Managing Architectural Technical Debt: A unified model and systematic literature review

The following sections will briefly describe Paper A. More details of this study are reported in Chapter 9.

8.1.1. Study Summary

The goal of this study was to synthesize and compile the current 'state-of-the-art' in the ATD field by conducting a systematic literature review focusing on the following research areas: ATD in terms of principal, interest, debt, and related challenges and solutions for managing ATD. Brereton et al. [90] state that software engineering systematic reviews can be categorized as being qualitative in nature, and, with this information as background, we conclude that our study is both qualitative and quantitative in nature, where the qualitative approach refers to the synthesizing process of the reviewed publication, and the quantitative approach refers to the mapping of the retrieved number of publications into the study's unified model. However, the approach used in this paper has a strong emphasis on a qualitative approach with less emphasis on a quantitative approach.

The study was conducted by automatically searching in six well-known digital libraries: the ACM Digital Library, IEEExplore, ScienceDirect, SpringerLink, Scopus, and Web of Science, as well as a manual hand search in all the proceedings of a key conference on the subject: the International Workshop on Managing Technical Debt (MTD Workshop). Additionally, we also conducted both a forward and backward snowballing technique to the retrieved publications.

The target of the search term was defined so as to search in both title and abstract, and the search term (query) contained the keywords: **"technical debt" AND architec∗.** The search was conducted in April 2017 and included publications between 2005–2016.

To screen the most interesting and relevant publications for this review, a filtering technique based on five different stages was used. In the first stage, 166 publications from the different data sources were retrieved and merged. The second stage of finding and

المنارة للاستشارات

www.r

removing duplicates resulted in reducing the number of publications to 79. The third stage was applied after the full texts were retrieved, where each publication was checked using the defined inclusion and exclusion criteria. This stage returned 38 publications, to which the snowballing technique was applied in the fourth stage. In stage five, we again applied the inclusion and exclusion criteria to the publications retrieved using the snowballing technique, which returned 42 publications for a detailed quality assessment. In the sixth stage, the publications went through an assessment process, with the goal of assessing the quality of all publications. Finally, in the seventh stage, data were extracted from each of the 42 included primary publications. Based on these 42 publications, we conducted a synthesis, including a descriptive synthesis and a quantitative summary (meta-analysis), by studying selected venues, numbers of publications per venue, publication types, and publications per year.

There was a wide agreement in the reviewed publications that ATD is of primary importance to software development. However, it was observed that ATD is described in a scattered and inconsistent manner. Consequently, we concluded that, in order to derive more value from the results concerning ATD and its effects, a holistic model depicting different views and their implications was required. We thus constructed a novel descriptive model that provided an overall understanding of existing knowledge in the research area of ATD, with the aim of providing a comprehensive interpretation of the ATD phenomenon. This model clarifies the different aspects of each research question and assembles relationships between them, together with an ATD identification checklist, recognized ATD impediments, and identified ATD management strategies.

8.1.2. Results (contributing to RQ1.1)

The main objective of this study was to elucidate and contribute to an extended knowledge base in the research area of ATD, and to build a collective platform for future research. The contributions of this study are both in the academic and practical aspects. First, the study shows that there is no single unified and overarching description or interpretation of ATD, and we, therefore, provide a 'state-of-the-art' review of significant issues which identifies aspects of previous studies and examines how these studies have been conducted. This study also provides a novel descriptive model that can support the process of more informed management of the software development lifecycle, with the goal of raising the system's success rate and lowering the rate of negative consequences for both the academic and practitioner community. This will allow practitioners and researchers to use this model to assess and recognize what problems might occur while dealing with ATD, as well as the consequences of these challenges being left unattended. This study also shows that there is a compelling need for supporting tools and methods for system monitoring and evaluation of ATD, but also shows that no software tools covering the full spectrum of ATD are currently available. Furthermore, the results demonstrate that maintenance and evolvability are the main challenges within ATD, due to the fact that all of the ATD challenges are related to compromises in these quality attributes. This paper highlights that practitioners in general lack strategies for architectural refactoring, and such activity might, therefore, result in an ad-hoc process where the results are inadequate. Consequently, this paper provides several key

المنارة للاستشارات
www.r

dimensions that need to be taken into consideration when defining a refactoring strategy for ATD issues. In addition, this study demonstrates, to both practitioners and academics, the relevance of dedicating more attention and effort to remediate ATD during the software lifecycle, in order to decrease the level of negative impact due to ATD on daily software development work.

Together with the findings in Paper D, this study answers the first sub-question of the first research question (RQ1.1) by presenting several negative effects of ATD, as illustrated in Figure 10. The negative impact of ATD can lead to severe and costly maintenance overheads which, in the long run, can diminish an organization's ability to innovate and evolve. ATD can also result in restricted flexibility, decreased reliability, and performance degradation.

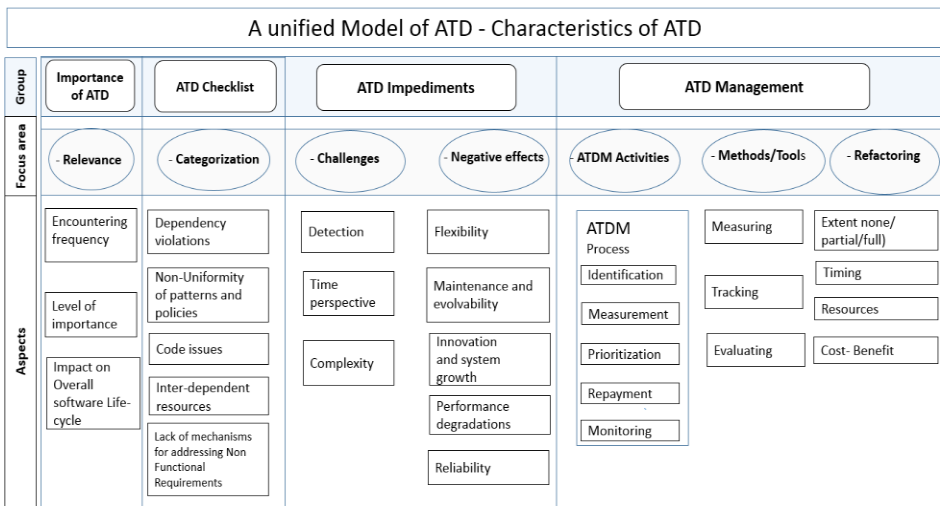


Figure 10: Characteristics of ATD, based on the results from Paper A.

8.2. Paper B: Time to Pay Up – Technical Debt from a Software Quality Perspective

The following sections will briefly describe Paper B. More details of this study are reported in Chapter 10.

8.2.1. Study Summary

The results of the SLR showed that, within the reviewed publications, the most frequently identified negative effects caused by ATD were compromises of maintainability and evolvability, which led to this second study, where we empirically investigated in which manner TD affects different software quality attributes.

The second sub-question of the first research question (RQ1.2) in this thesis addresses how TD affects different software quality attributes. In order to be able to answer RQ1.2, we conducted a study with the aim of understanding which quality issues have the most

المنارة للاستشارات
www.r

negative impact on the software development lifecycle process, and to determine the association of these quality issues in relation to the age of the software, as well as to relate each of these quality issues to the impact of different TD types. This study was conducted through a combination of qualitative and quantitative research approaches and was conducted in three different stages.

First, we group-interviewed 43 software practitioners, with the goal of understanding which of the quality attributes were the most negatively affected by TD. This stage also included an assessment of compromised quality attributes by an in-depth analysis, examining nine different TD issues, and evaluating the impact each of these had on different quality attributes listed by ISO/IEC. In the second stage, an online survey was used, providing quantitative data from 258 software participants. In the third stage, we conducted seven semi-structured follow-up group interviews with a total of 32 industrial software practitioners.

8.2.2. Results (contributing to RQ1.2)

The second sub-question of the first research question (RQ1.2) addresses how TD affects different software quality attributes, which is addressed in this paper. This paper provides an empirically based study on how practitioners experience and perceive TD in terms of compromised quality attributes and their relation to wasted working time, based on both quantitative and qualitative data.

First, the results of this study show that practitioners identified *maintenance difficulties*, a *limited ability to add new features*, *restricted reusability*, *poor reliability*, and *performance degradation* issues as the quality attributes with the most negative effect on the software development lifecycle. When analyzing these five quality attributes, the summary statistics from the survey showed that 60% of all respondents frequently or very frequently encounter maintenance difficulties during the software lifecycle, 45% of the respondents frequently or very frequently encounter restricted reusability, and 39% of the respondents frequently or very frequently encounter a limited ability to add new features.

Secondly, we found no evidence for the generally held opinion that maintenance complications increase with the age of software. When studying how the different examined quality attributes were compromised with respect to the age of the software, our results showed that there were only significant differences in how the respondents encountered maintenance difficulties regarding its age. Furthermore, our results showed that, for software older than 20 years and for systems within the age interval of 2–10 years, the respondents encounter maintenance difficulties most frequently, and, for systems within the interval of 10–20 years, a limited ability to add new features is the most frequently encountered problem. Respondents who reported restricted reusability as the most frequently encountered quality issue had software with an average age of less than two years. This study could thus not confirm the generally held view that the amount of compromised quality attributes increases with system age, but our results imply that it is important to remediate TD very early in the lifecycle in order to keep the frequency of compromised quality attributes down.

المنارة للاستشارات

www.r

Thirdly, we show that TD affects not only software productivity (in terms of wasted time) but also that several quality attributes of the system were negatively affected by TD. The results showed that there is a significant positive linear correlation between the frequency of encountering all of the investigated quality issues and the estimated amount of wasted time. The strongest relationship was found between the frequency of encountering poor reliability and wasted time, followed by a limited ability to add new features and maintenance difficulties.

These findings highlight the importance of understanding how TD negatively affects overall system quality in order to proactively manage it in terms of allocating time, resources, and additional effort. These findings provide strong empirical confirmation that both practitioners and academics need to focus more attention and effort on deliberately remediating TD, in order to reduce future costly interest payments.

8.3. Paper C: The Pricey Bill of Technical Debt – When and by Whom Will it be Paid?

The following sections will briefly describe Paper C. More details of this study are reported in Chapter 11.

8.3.1. Study Summary

From Paper A, it was evident that limited knowledge and few supporting tools are available to measure the extent of TD within a software application, and, in addition, the time spent on TD related issues is not made explicitly visible and measurable. The lack of this knowledge can result in the fact that software development organizations are not aware of the interest that they are paying on TD, and therefore they might not give TD management the necessary attention.

The third sub-question of the first research question (RQ1.3) addresses the negative impact on development productivity due to TD. This question is answered in two studies (Papers C and E), both exploring the amount of waste of working time due to TD. This research study also addresses RQ2.1 and RQ2.2, focusing on the impact of TD remediation and prevention activities and its impact on reducing the negative effects of TD.

This first study's goal was to empirically investigate how software practitioners *perceive* and *estimate* the interest payment of TD. More specifically, the main goal of the study was to examine the amount of estimated wasted time caused by TD interest during the software lifecycle. The aim of this study was also to investigate what type of TD generates the most negative effects and the activities on which the extra time was spent as a result of TD. Furthermore, this study also examined the ways in which the age of the software system affected the wasted time, the frequency of encountering different TD types, and the frequency of having to perform the extra activities. Finally, this study also investigates the ways in which different professional software roles are affected by these artifacts.

To accomplish this goal, we conducted a study using a combination of qualitative and quantitative research approaches. First, an online survey was sent to seven companies

المنارة للاستشارات

www.r

within our networks with an extensive range of software development, and invitations were also published in software engineering-related networks on LinkedIn. In total, this survey returned 258 complete answers (completion rate of 83%). Secondly, we conducted follow-up interviews with 32 industrial software practitioners, who had all participated in the first survey. During the semi-structured interviews, the compiled results from the previous survey were presented to the interviewees, where some of the most interesting findings were highlighted together with questions related to the specific area of research. This presentation allowed the interviewees to more easily relate the interview questions to the results of the survey.

8.3.2. Results (contributing to RQ1.3, RQ2.1, and RQ2.2)

The third sub-question of the first research question (RQ1.3) attempts to explore the negative impact on development productivity due to TD, which is addressed in Papers C and E. In both these studies, we have used the amount of wasted time as a proxy for *productivity*.

This study offers a contribution to TD research, with respect to the existing body of knowledge, in several respects. The single most striking result emerging from this study is that, on average, software practitioners *estimate* that 36% of all software development time is wasted due to paying the interest of TD. The result reveals that the majority of the wasted time is spent on understanding and/or measuring TD. When studying how different professional roles perceive TD, this result reveals that different roles are affected differently by TD. We found that different roles waste time on different activities, hence experiencing different negative impacts of TD. When examining whether, and in what way, the amount of perceived wasted time varies with respect to the age of the software, this study shows that the degree of the wasted time does vary. Although the amount of wasted time result does not show a linear progression, the result shows that wasted time varies in relation to the system's age.

Further, the findings of the second research questions (RQ2.1 and RQ2.2) focusing on prevention and remediation activities, in order to reduce the negative effects of TD, indicate that organizations can lower the amount of waste of working time (and thereby increase the productivity) by continuously preventing the introduction of TD in the first place and reducing the amount of TD by remediation activities, which means that software companies could benefit from raising their awareness about the amount of time and resources they are spending on TD, and to deliberately focus on preventing and remediating their TD.

8.4. Paper D: Impact of Architectural Technical Debt on Daily Software Development Work – A Survey of Software Practitioners

The following sections will briefly describe Paper D. More details of this study are reported in Chapter 12.

المنارة للاستشارات

www.r

8.4.1. Study Summary

An important and interesting result from the previous study (Paper C) showed that when software practitioners estimate the negative impact of several different types of TD, they perceive that ATD generates the most negative impact on their daily software development work (closely followed by Requirement TD).

The first sub-question of the first research question (RQ1.1) addresses the negative impact of *architectural* TD. Based on the previous results from Paper A, in Paper C we conducted a more in-depth analysis of the *architectural* related aspects of TD.

The goal of this study was to examine how software practitioners perceive and estimate the impact of specifically ATD during the software development process from several different aspects. The first goal set out to examine the level of negative effects ATD has on the daily software development work and compare this level with the negative effects of other types of TD. Secondly, we aimed to understand whether the level of the negative effects due to ATD correlates with the estimated wasted development time during the software lifecycle. Thirdly, the negative effects due to ATD is commonly believed to have an increasingly negative impact with respect to the age of the software. Finally, this study aimed to assess the extent to which the level of negative effects due to ATD differs in relation to the age of the system and whether different professional roles are affected differently by specific ATD.

The data in this study were collected via surveys. This paper is to some extent also related to our previous paper, Paper C, wherein we study and compare several different TD types. However, even if the data are collected using the same survey, this study focuses on the architectural aspects of TD and does not focus on other types of TD besides ATD. By focusing specifically on ATD, this means that we can provide a more in-depth analysis of the architecturally related issues of TD as well as a more detailed statistical analysis of the data.

8.4.2. Results (contributing to RQ1.1)

The first sub-question of the first research question (RQ1.1) concentrates on the negative impact of *architectural* TD. This question is answered in Papers A and D, where in Paper D we study the negative impact ATD has on the overall daily software development work, with respect to wasted working time and additional activities performed.

First, the results of this study show that practitioners experience that ATD has the highest negative impact on daily software development work. The results of the study also show that the level of negative impact due to ATD is introduced early, and thereafter remains during the entire software lifecycle. Based on evidence from our survey, this study does not support the currently held belief that the negative effects due to ATD increase with respect to the age of the system. This study also provides new insights into ATD research by showing that, despite the different responsibilities and working tasks of software professionals, ATD negatively affects all roles without any significant difference between these roles. This study contributes to an empirical confirmation that software companies

المنارة للاستشارات
www.m

need to invest in continuous refactoring from the conception of the system in order to maintain the negative effect generated by ATD at a future low level.

8.5. Paper E: Software Developer Productivity Loss Due to Technical Debt – A replication and extension study examining developers' development work

The following sections will briefly describe Paper E. More details of this study are reported in Chapter 13.

8.5.1. Study Summary

In the previously presented paper, Paper C, the results show that software engineers *estimate* that, on average, they waste 36% of their software development time due to TD. Even if the respondents in that study were experienced in software development, and their estimates were likely to be formed by what they have heard, observed, and experienced at their workplaces, we were intrigued by the idea of conducting an additional study where the wasted time could be studied by using *reported* data instead of a single occurrence based on *perception* and *estimates*. In order to further answer RQ1.3, we extended the previous research exploration by incorporating a longitudinal study where software developers *reported* their experience and interest due to TD (*instead of using single estimates*) twice a week for seven weeks. The goal of this study was to explore the negative consequences of TD in terms of wasted software development time as a proxy for productivity. This study also investigates on which additional activities this wasted time was spent and whether different types of TD impacted the wasted time differently. It also set out to examine the benefits of tracking and communicating the amount of wasted time, from both a developer's and manager's perspective. This study reports the results of a longitudinal study surveying 43 developers (473 data points) and includes 16 interviews, followed by a validation by an additional study using a different and independent dataset (n = 47, 177 data points) focusing on replicating the findings. This research study also addresses RQ2, focusing on the impact of TD remediation and prevention activities.

8.5.2. Results (contributing to RQ1.3, RQ2.1, and RQ2.2)

The third sub-question of the first research question (RQ1.3) concentrates on the amount of wasted software development time, as a proxy for software development productivity.

This study makes a novel contribution to the TD research, where the analysis of the reported wasted time revealed that *developers* report that they waste, on average, 23% of their software development time due to TD and that the wasted time is most commonly spent on performing additional testing, followed by conducting additional source code analysis and performing additional refactoring. The results also reveal that, on a quarter of the occasions where developers encounter TD, they are forced to introduce additional TD due to the already existing TD. The results from the replication study were, overall,

المنارة للاستشارات
www.r

in line with the results from the original study regarding the amount of waste of working time and the ways different technical debt types affect the wastage.

By studying the tracking process of the wasted time, it was apparent that none of the examined companies tracked or measured the amount of wasted time due to TD, and none of the companies had an aligned strategy for addressing the interest of TD. In addition, this study shows that both developers and managers clearly see the benefits of tracking the amount of wasted time, yet both professions are somewhat reluctant to implement such measures in practice. This "unwillingness" is recognized as a challenge by the companies.

Further, the findings of the second research question (RQ2) show that, in a quarter of the occasions where developers encounter TD, they are forced to introduce additional TD due to the already existing TD. This burden of being forced to introduce additional TD demonstrates the contagiousness of TD, and our results suggest that TD should be prioritized for refactoring because it forces the developers to introduce further additional TD, which generates even more interest. This finding shows that conducting both prevention and remediation activities of TD, with the goal of reducing the amount of TD, will also reduce the waste of working time and thereby increase overall developer productivity.

8.6. Paper F: The Influence of Technical Debt on Software Developer Morale

The following sections will briefly describe Paper F. More details of this study are reported in Chapter 14.

8.6.1. Study Summary

The previously described studies in Papers A, B, C, D, and E demonstrate the negative consequences of TD and ATD, from both a technical and an economic perspective. However, TD can also affect developers' psychological states and morale, which is the primary focus in RQ1.4. This research study also addresses RQ2.2, focusing on the impact of TD remediation activities.

Drawing on the previous literature on morale, this study explores the influence of TD and its management on three dimensions of morale: affective, future/goal, and interpersonal antecedents. Furthermore, in order to understand whether their morale affects the developer's work productivity, this study explores associations between morale and the amount of reported wastage of working time due to experiencing TD. In this study, we followed a mixed-methods approach to both quantitative and qualitative research. The quantitative approach was performed through the first survey with 33 software developers followed by a second survey with 43 developers (collecting 473 data points), and the qualitative part of the study was conducted through 15 semi-structured interviews.

المنارة للاستشارات

www.r

8.6.2. Results (contributing to RQ1.4, and RQ2.2)

The fourth sub-question of the first research question (RQ1.4) investigates the negative impact TD has on developers' *morale* and is addressed in Paper F.

This study has several contributions to both software engineering research and practice. The study specifically concentrates on investigating the influence of TD on developers' morale and its correlation to productivity. While the study is based on previous literature on morale, the findings from this study make several contributions to the present TD research.

The results from this study indicate that the occurrence of TD reduces developers' morale since the presence of TD hinders the developers' progress and reduces their confidence. Further, the results imply that proper management of TD increases developers' morale since it enables developers to perform their tasks better and to improve software quality in the future.

Further, the findings of the second research question (RQ2.2) in this study suggest that proper TD management can lead to a virtuous cycle where the right culture and TD prevention mechanisms reinforce each other, leading to less waste of time, followed by a continuous increase of the developers' morale and productivity.

8.7. Paper G: Technical Debt Management: Current State of Practice

The following sections will briefly describe Paper G. More details of this study are reported in Chapter 15.

8.7.1. Study Summary

As previously shown in all the publications mentioned above, TD has a negative impact on software development from various different perspectives, and the results from these publications demonstrate the relevance of paying more attention and effort to actively managing and remediating TD. When implementing a TD management strategy, the *tracking* of the TD is an important key activity. Therefore, this thesis' research question (RQ2.3) focuses on the impact of TD tracking initiatives on overall software development.

This study was conducted using both qualitative and quantitative methods. First, we conducted a survey of 226 respondents from 15 organizations and followed up with multiple case studies at three companies that have started tracking TD. The case study included 13 semi-structured interviews and a collection of 79 TD-related documents.

8.7.2. Results (contributing to RQ2.3)

The third sub-question of the second research question (RQ2.3) examines the impact of TD tracking processes and is addressed in both this paper and in Paper J. The results from this study show that software practitioners estimate that, on average, they spend a

المنارة للاستشارات
www.m

substantial amount of their working time trying to manage TD and only a few of them have started tracking TD, where 7.2% of them apply a systematic tracking process in this regard. The results further show that the major reasons for this noticeably low proportion of companies having an implemented TD tracking process are due to lack of knowledge of what is necessary to implement it in terms of tools and processes, as well as a lack of awareness of what the negative effects of TD are before they occur. In order to help the initialization process for TD tracking, we propose a Strategic Adoption Model (SAMTTD). This model can be used by practitioners to assess their TD tracking process and to plan their next steps.

8.8. Paper H: Embracing Technical Debt, from a Startup Company Perspective

The following sections will briefly describe Paper H. More details of this study are reported in Chapter 16.

8.8.1. Study Summary

Today, most research on TD has been focused on mature software teams and organizations, who may have less pressure to position new and innovative software quickly in a new market and, therefore, reason about TD quite differently than software startups, for example. In this study, we seek to understand the organizational factors that lead to, and the benefits and challenges associated with, the intentional accumulation of TD in software startups. In this study, we interviewed 16 professionals involved in seven different software startups.

The first sub-question of the third research question (RQ3.1) focuses on the consequences of TD in context-specific domains and, more specifically, addressing the challenges and benefits of deliberately introducing TD for software startups.

8.8.2. Results (contributing to RQ3.1)

Our results show that software startups differ significantly from mature organizations in how they accumulate and manage TD. We find that the startup phase, the experience of the developers, software knowledge of the founders, and the level of employee growth are some of the organizational factors that influence the intentional accumulation of TD. In addition, we find that software startups are typically driven to achieve a “good enough” level, and this guides the amount of TD that they intentionally accumulate to balance the benefits of speed to market and reduced resources with the challenges of later addressing TD. This study also presents several benefits of intentionally taking on TD, where these kinds of strategic decisions are taken by practitioners being relatively aware of the harmful effects these decisions can have on the future software in terms of impeding innovation and expansion of their software systems. Further, this study also provides a set of recommendations and a first strategy that can be used by software startups to support their decisions related to the accumulation and refactoring of TD.

المنارة للاستشارات

www.r

8.9. Paper I: How Regulations of Safety-Critical Software Affect Technical Debt

The following sections will briefly describe Paper I. More details of this study are reported in Chapter 17.

8.9.1. Study Summary

In today's software industry, the use of safety-critical software (SCS) is increasing at a rapid rate. However, little is known about the relationship between the safety-critical regulations in this type of software and TD. While there are several similarities between non-SCS and SCS, there are also several major differences. One of the most significant differences is that SCS is heavily regulated and requires certification against industry standards. These standards have, among other issues, an adverse impact on the process of conducting refactoring tasks. This study focuses primarily on the consequences and effects of SCS on TD, the factors influencing the decision-making of TD refactoring, and also what software architectural structures contribute to TD refactorings in SCS. In this study, we interviewed nine practitioners working in different safety-critical domains implementing software according to different safety regulation standards.

This research study addresses RQ2.1 and RQ2.2, focusing on the impact of TD remediation and prevention activities in order to reduce the negative effects of TD. Further, this study also addresses the second sub-question of the third research question (RQ3.2) by focusing on the consequences of TD in context-specific domains, more specifically the impact regulatory requirements have on TD Management in a safety-critical software context.

8.9.2. Results (contributing to RQ2.1, RQ2.2, and RQ3.2)

The results of this study show that performing TD refactoring tasks in SCS requires several additional activities and costs, compared to non-safety-critical software.

The findings of the research questions RQ2.1 and RQ3.2 show that the SCS regulations strengthen the implementation of both source code and architecture and thereby initially limit the introduction of TD. However, at the same time, the regulations also force software companies to perform later suboptimal workaround solutions that are counterproductive to achieving high-quality software since the regulations constrain the possibility of performing optimal TD refactoring activities.

Further, when addressing RQ2.2, the result shows that SCS regulations have an immediate and adverse impact on the refactoring and remediation activities of TD. Due to additional costs and the need for additional activities due to the regulations, practitioners frequently avoid performing refactoring activities, with the consequence of introducing additional TD in the form of, for example, suboptimal workaround solutions.

المنارة للاستشارات

www.m

8.10. Paper J: Technical debt triage in backlog management

The following sections will briefly describe Paper J. More details of this study are reported in Chapter 18.

8.10.1. Study Summary

Remediation of TD through regular refactoring activities in the software are considered vital for the software system's long and healthy life. However, to select the TD tasks that should be refactored, one must first prioritize the different identified TD tasks. Therefore, this study's goal is to explore how the prioritization of TD is carried out by practitioners within today's software industry and also to investigate which factors influence the prioritization process and its related benefits and challenges. This paper reports the results of surveying 17 software practitioners, together with follow-up interviews. The second sub-question of the second research question (RQ2.2) focuses on the impact TD remediation initiatives have on software development, and this question is partly answered in this paper. This paper also addresses RQ2.3 by focusing on how the TD items are tracked in, e.g., backlogs.

8.10.2. Results (contributing to RQ2.2, and RQ2.3)

This study's results show that there is no uniform way of prioritizing TD and that it is commonly done reactively without applying any explicit strategies. Further, the results show that often the TD issues are managed and prioritized in a shadow backlog, separate from the official sprint backlog. Even if the TD items sometimes are escalated by the transition from the shadow backlog into the main backlog, this transition is not clear in terms of when and if the TD item should be transferred. This way of administrating TD items could potentially lead to the TD items becoming not fully visible and known during the prioritization process, and thereby, not given sufficient attention.

Further, the second sub-question in the second research question (RQ2.2) focuses on the impact of remediation activities, and the result of this study shows that the process regarding which TD item to prioritize is heavily influenced by gut feelings among the decision-makers, and whether the user of the software is an external or internal character also influences the prioritization strategy of TD. Further, when answering RQ2.3, the results indicate that TD items are not sufficiently visible and tracked in the official backlogs and thereby not prioritized accordingly.

8.11. Paper K: Carrot and Stick approaches when managing Technical Debt

The following sections will briefly describe Paper K. More details of this study are reported in Chapter 19.

المنارة للاستشارات
www.r

8.11.1. Study Summary

Even if practitioners working in today's software development industry are quite familiar with the concept of TD and its related negative consequences, there has been no empirical research focusing specifically on how software managers actively communicate and manage the need to keep the level of TD as low as possible. Similar to other professionals, software engineers' work outcomes, attitudes, and work behaviors are influenced by a company's corporate culture together with the managers' mindset. This means that managers can have a very large impact on the overall software development process by adopting different management strategies and using techniques for controlling and directing software engineers to achieve pre-determined goals.

The second sub-question of the second research question (RQ2.2) focuses on remediation activities and is partly answered in this paper by studying how managers can influence practitioners addressing TD, where this study explores how software companies encourage and reward practitioners for actively keeping the level of TD down, and also whether the companies use any forcing or penalizing initiatives when managing TD. The research questions in this study focus on how common each of those four strategies and whether software engineering practitioners perceive these TD management strategies as effective or desirable. This paper reports the results of both an online survey providing quantitative data from 258 participants and follow-up interviews with 32 industrial software practitioners.

8.11.2. Results (contributing to RQ2.2)

The results show that having a TD management strategy (based on the four assessed strategies) can significantly impact the amount of TD in the software by adopting, e.g., different remediation initiatives.

When surveying how commonly different TD management strategies are used, we found that only the encouraging strategy is, to some extent, adopted in today's' software industry. Further, this result shows that there is an unfulfilled potential for managers to impact how practitioners can reduce and remediate TD by adopting a TD management strategy based on encouragement. This study also provides a model describing the four assessed strategies by presenting strategies and tactics, together with recommendations on how they could be operationalized in today's software companies.

المنارة للاستشارات

www.m

9. Managing Architectural Technical Debt

In this chapter, we synthesize and compile research efforts with the goal of creating new knowledge with a specific interest in the Architectural TD (ATD) field.

Large Software Companies need to support the continuous and fast delivery of customer value in both the short and long term. However, this can be impeded if the evolution and maintenance of existing systems are hampered by what has been recently termed Technical Debt (TD). Specifically, ATD has received increased attention in the last few years due to its significant impact on system success and, left unchecked, it can cause expensive repercussions. It is therefore important to understand the underlying factors of ATD. With this as background, there is a need for a descriptive model to illustrate and explain different ATD issues. The contribution of this paper is the presentation of a novel descriptive model, providing a comprehensive interpretation of the architectural TD phenomenon. This model categorizes the main characteristics of ATD and reveals their relations. The results show that, by using this model, different stakeholders could increase the system's success rate, and lower the rate of negative consequences, by raising awareness about ATD.

This chapter has been published as:
Managing architectural technical debt: A unified model and systematic literature review
Terese Besker, Antonio Martini, and Jan Bosch,
Journal of Systems and Software, vol. 135, pp. 1-16, 2018

9.1. Introduction

In 1992, Ward Cunningham [91] introduced the financial metaphor of Technical Debt (TD) to describe the need for recognizing the potential long-term and far-reaching negative effects of immature code, made during the software development lifecycle, which has to be repaid with interest in the long term. Cunningham used the financial terms of debt and interest when describing TD: "*Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt*." Another, more recent, a definition was provided by Avgeriou et al. [4] who define TD as "*In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*."

Today, large-scale software companies strive to increase their efficiency in each lifecycle phase, by reducing time and resources deployed by the development teams. To achieve this goal of delivering high-quality systems, software architecture is especially important

المنارة للاستشارات
www.m

and should contribute to a minimal maintenance effort. [92] states that maintenance activities consume 50-70% of the total effort spent during a typical software project. Left unchecked, these maintenance activities can make the architecture diverge towards a suboptimal state, and, in the worst-case scenario, towards system obsolescence or crisis [93]. During the development of large-scale systems, software architecture plays a significant important role [18], and consequently, a vital part of the overall TD relates to sub-optimal architectural decisions and is regarded as Architectural Technical Debt (ATD) [94], [23]. ATD is primarily incurred by architectural decisions with the consequence of immature architectural artifacts and compromised quality attributes [95]. ATD commonly refers to violations of best practices [93], consistency and integrity constraints of the architectures, or the implementation of immature architecture techniques. These consequences primarily result from the compromise of modularity, reusability, analyzability, modifiability, testability, or evolvability during software architecting [96]. ATD does not always receive the full attention from the architect and management teams due to the fact that ATD typically concerns the cost of the long-term maintenance and evolution of a software system instead of the visible short-term business value. Another reason for this is that ATD is hard to identify and measure since it is not easily visible [28] within the codebase. The visibility of ATD, generally first occurs when the system significantly reveals shortcomings or complications in the maintenance or operation [22]. TD management involves navigating a path that considers both value and cost, to focus on overall ROI over the software lifecycle [97] for all different types of TD, and, although a great deal of theoretical work on the architectural aspects of TD has recently been produced, practical ATD Management (ATDM), with an architectural focus, lacks empirical studies [98].

ATD is evidently detrimental to software companies, and since ATD has such a significant negative impact on software systems [95],[94], it is important to understand what it is and of what it is composed. This study focuses on different ATD categories and their related effects and thereby becomes highly relevant when providing a platform for analyzing different ATDM strategies, solutions, and challenges. To date, we have not found any studies describing this issue in a unified way, which could facilitate the challenges of understanding and manage ATD in an overall context. Research can benefit from research synthesis techniques that help summarize and assess the body of results accumulating in the literature [99]. Therefore, to explore and understand these concerns in a more comprehensive context, a systematic literature review is conducted in the area of ATD, with research questions focusing on the current knowledge regarding debt, interest, principal and existing challenges, and solutions in managing ATD.

This unified model and literature review can be of benefit from a variety of academic perspectives. For researchers interested in the architectural aspects of TD, the research agenda for ATD helps to build upon already existing work and guide efforts towards new research directions. For practitioners, the unified model can help to identify ATD and to evaluate what problems might occur while dealing with ATD, and the consequences of these challenges are left unattended.

The objective of this study has been achieved by employing a thorough Systematic Literature Review (SLR) research method [100]. SLR is a well-established research

المنارة للاستشارات

www.r

method for conducting a structured and systematic way of performing a review using a protocol by formally defining each process within the review process.

The main objective of this study is to clarify and contribute to an extended knowledge base in the research area of ATD and to create a common platform for future research. The contributions of this study are as follows:

1. We present results showing that there is no one unified and overarching description or interpretation for ATD and, therefore, a 'state-of-the-art' review of significant issues is provided, concerning various ATD issues.
2. This study identifies aspects of previous studies and examines how the studies have been conducted.
3. This study provides a novel descriptive model that provides an overall understanding concerning knowledge currently of interest in the research area of ATD. This unified descriptive model can support the process of more informed management of the software development lifecycle, with the goal of raising the system's success rate and lowering the rate of negative consequences for both the academic and practitioner community.
4. Having this visual and unified model in which stakeholders can rapidly obtain a holistic overview is valuable. Researchers and practitioners can use this unified model to evaluate and understand what problems might occur while dealing with ATD and the consequences if these challenges are left unattended. In our unified model, we provide a checklist of the aspects of ATD reported in the literature. Researchers and practitioners can use this checklist as a general reference tool for recognizing ATD.
5. This study shows that there is a compelling need for supporting tools and methods for system monitoring and evaluating ATD, but also shows that no software tools covering the full spectrum of ATD are yet available.
6. This study provides new insights into the refactoring of ATD research by showing that practitioners, in general, lack strategies for architectural refactoring, and, therefore, such activity might result in an ad-hoc process where the results are inadequate. In this paper, we provide the key dimensions that need to be taken into consideration when defining such a refactoring strategy.
7. To both practitioners and academics, this study demonstrates the relevance of paying more attention and effort to remediate ATD during the software lifecycle, in order to decrease the level of negative impact due to ATD on daily software development work.

This paper is structured into nine sections. The following section introduces background information that is used during a discussion of the results. In the third section, the SLR method is described. The fourth section presents the results from the retrieval of publications, and section five presents the results of the data collection of the publications. Section six addresses the importance of ATD and the need for a unified model. Section seven discusses the results of both the literature review and the unified model and

المنارة للاستشارات
www.r

analyzes the results of an ATD research agenda and lists the threats to the validity. Section eight reviews the related research, while the last section, nine, concludes the paper.

This paper is an extension of the previously published paper that was originally published at the 42nd Euromicro Conference on Software Engineering and Advanced Applications (SEAA) in 2016 [101]. However, this paper includes several more surveyed publications, since the timeframe when searching for publications was expanded to also include the year 2016. We have also added both a forward and backward snowballing technique for this extra time period. This paper also includes an additional research question, and, consequently, new findings and a more in-depth analysis of all of the research questions have been obtained. Moreover, the unified model offered is an extension of the previous model presented in the abovementioned conference paper [101], where the model has been enhanced by including an additional research question (importance of ATD - RQ1) and updated research results for all of the different aspects of the model.

9.2. Background

In order to provide the reader with the necessary information that is needed to better understand the remainder of the paper, this section provides a background to the ATD domain. In this broad view, we examine what constitutes ATD in terms of debt, interest, and principal and how it is managed with regard to related management processes, current challenges, and analyzing support.

There are different types of TD, with Alves et al. [14] having identified and organized the different types by considering the nature as a classification criterion for each TD type. They identified 13 different types of TD, including Architectural TD, Build Debt, Infrastructure TD, Requirement TD, Test Automation TD, and Code TD. Alves et al. define ATD as referring "to the problems encountered in project architecture, for example, violation of modularity, which can affect architectural requirements (performance, robustness, among others). Normally this type of debt cannot be paid with simple interventions in the code, implying in more extensive development activities." In a similar manner, Fernández-Sánchez et al. [24] describe ATD as being caused by shortcuts and shortcomings in design and architecture or by the result of sub-optimal upfront architecture design solutions, that become sub-optimal as technologies and patterns become superseded.

9.2.1. Debt

In financial terms, a debt refers to the amount of money owed by one party (debtor or borrower) to another party (creditor or lender) [30], where the obligation of the debtor is to repay a larger sum of money to the creditor at the end of that period [31]. From a software development perspective, the same term is used to describe the gap between the existing state of a software and some hypothesized "ideal" state in which the system is optimally successful [32].

[19] and [20] describe that taking on debt does not always arise from recklessness, TD can also be created deliberately and inadvertently due to strategic decisions. ATD is

المنارة للاستشارات

www.r

commonly caused by architectural decisions (conscious or unconscious) that compromise system-wide quality attributes, especially maintainability and evolvability [22] and can be seen as expedient in the short term but can be more costly in the long term. These shortcomings can be summarized as ATD, or architectural debt, in the sense that they are system defects that can be improved to form an enhanced architectural software quality and to avoid excessive interest payments in the form of decreasing maintainability.

9.2.2. Interest

Interest is the most commonly used financial term in TD research, and Ampatzoglou et al. [30] define interest in their TD financial glossary list as: "The additional effort that is needed to be spent on maintaining the software, because of its decayed design-time quality." Interest is the negative effects of the extra effort that has to be paid due to the accumulated amount of debt in the system, such as executing manual processes that could potentially be automated, excessive effort spent on modifying unnecessarily complex code, performance problems due to lower resource usage by inefficient code, and similar costs [7], [25]. It is vital that the elements and structures of ATD are regularly monitored during the software lifecycle to support an analysis of how debt is building up with interest.

9.2.3. Principal

In financial terms, principal refers to the original amount of money borrowed, and, from a software development perspective, the same term is used to describe the cost of remediating planned software system violations concerning TD, in other words, the cost of refactoring. Ampatzoglou et al. [30] define principal within a TD context as: "The effort that is required to address the difference between the current and the optimal level of design-time quality, in an immature software artifact or the complete software system." The aggregate of the principal can be computed as a combination of the number of violations in an application, the hours to correct each violation and the cost of labor [102]. With this type of informative background analysis, the management can decide when, and how, to perform the refactoring of the ATD by deploying a refactoring strategy.

9.2.4. Software management process

The objectives of different software management activities performed during the ATD Management process include the main activities of: identification, measurement, prioritization, repayment, and monitoring [22]. The identifying activity includes locating the ATD, identifying the causes, the compromised quality attributes, and the impact on future development [103]. One of the most challenging activities during the ATDM is the measuring of the quantitative characterization of the system-specific ATD items [32], where designing and validating different measurement approaches are crucial for the quality and reliability of this activity output. The prioritization activity should prioritize the ATD items due to an in-depth assessment of the system's business goals and preferences. The prioritization is intended to reflect factors such as cost and technical

المنارة للاستشارات

www.r

issues, regarding financial and technical impacts on the system. Repayment refers to making new or changing existing architecture decisions [22] in order to reduce or mitigate the undesirable effects of ATD in the system. The last activity within the process is a continuous integrated monitoring activity. This activity is based on an explicit and consequently controlling feature, with the mission of monitoring the level of ATD over time, identifying trends, and disseminating warnings at appropriate time intervals [32].

9.2.5. ATD challenges

One of the most challenging tasks encountered in the synthesis of ATD is how to translate architectural complications or debt into economic consequences, by means of predicting financial implications based on estimated values. A second key challenge is the estimation that arrives at a strict probability without evidence-based historical data that can provide an accurate estimate at the outset. Another challenge can be described in terms of the principal costs (e.g. refactoring of an ATD component) being difficult to estimate regarding the required working time, and Falessi et al. [104] state that the estimation of principal costs can, on average, result in a large, +/- 80%, of the most probable value. A key challenge during the estimation of interest can be explained by the implications of the type of interest curve in relation to the time aspect. The interest can be characterized by a curvature with a linear or nonlinear regression, which impedes the opportunities for accurate estimation. A reason for interest becoming non-linear can be explained due to contagion, caused by contaminants in other parts of the system with the same problem [23].

9.2.6. ATD analyzing support

Despite the significant need for supporting tools and methods for analyzing ATD, most of the available tools for analyzing TD focus on code level instead of the architectural aspects [27]. These code-focusing tools generally cannot provide indicative information for architectural trade-off since they can cause misleading results [41] and Nord et al. [97] describe that the existing metrics for system quality visibility are insufficient and unproven.

9.3. SLR method

The following section describes the method used for conducting this SLR. The SLR technique originates from medical research and has recently been adapted to software engineering. SLR is a secondary study method, meaning a study where the results from several primary studies are collected. The goal of a secondary study is to provide both researchers and practitioners with an overview of a specific field and also to identify gaps in the available literature [105].

This review procedure is based on guidelines by an established SLR method, described by [80]. The main rationale for undertaking a systematic review is to synthesize existing work, and that the review should be carried out in accordance with a predefined search strategy, which allows the search to be evaluated. The major advantage of using this

المنارة للاستشارات

www.r

method is that the result is provided by evidence, which is robust and transferable and that sources of variation can be further studied [80]. A review protocol was used to ensure rigor and repeatability involving the phases: a) define research questions, b) define search process, c) define inclusion and exclusion criteria, d) define quality assessments, e) define data collection, f) define data analysis, and g) define deviations from protocol.

9.3.1. Research question

As mentioned in Section I, the goal of this review is to obtain further knowledge on ATD regarding its constituents (debt, interest, and principal) and its management (challenges and solutions). Therefore, we will now define the following three main research questions (RQs) and the six related sub-questions. Each sub-question is related to an aspect of ATD defined in Section II.

RQ1: What is the importance of ATD in software development?

RQ2: What is the existing knowledge concerning ATD in terms of debt, interest, and principal?

RQ2.1: What are the categories of ATD?

RQ2.2: What are the major negative effects caused by ATD?

RQ2.3: What refactoring strategies are mentioned for managing ATD?

RQ3: What are the existing challenges and solutions for managing ATD?

RQ3.1: Which ATDM activities are mentioned in the literature?

RQ3.2: Are there any specific challenges with ATD?

RQ3.3: Are there any analysis methods for detecting and/or evaluating ATD?

The first research question, RQ1, seeks to address the relevance and importance of ATD to software development. The second research question, RQ2, concerns the existing knowledge on debt, interest, and principal, where "existing knowledge" refers to information captured from the retrieved publications. The first sub-question, RQ2.1, focuses on different categories of ATD that will provide information on how debt is described in the publications. Studying the outlined positive and negative effects caused by ATD in RQ2.2 will provide an understanding of the effects relating to different quality attributes (in terms of negative effects), which will be synthesized as interest. Question RQ2.3 focuses on the refactoring strategies revealed in publications, and this will be linked to the principal. The third research question, RQ3, aims at investigating challenges and solutions for managing ATD. This question is divided into three different sub-questions in order to increase the question's focus. The results obtained from studying which different ATD Management activities are stated in the reviewed publications by the utilization of results of RQ3.1 will reveal which activities are used as notions in the publications. RQ3.2 focuses on different kinds of challenges to the management of ATD

المنارة للاستشارات

www.m

and RQ3.3 concentrates on supporting activities for detecting and evaluating the software architecture.

9.3.2. Search process

A searching strategy process should include: (1) defining a searching term (query), (2) defining the target for the searching term, and (3) selecting different data sources with the aim of identifying candidate publications [80].

Since "Architectural Technical Debt" is not a common expression in the title or the abstract as a concatenated word sequence, publications that both contain the words "technical debt" and architec* were studied.

The *search term* (query) contains the following keywords:

"technical debt" AND architec*

The searching terms were combined using a Boolean AND operator, which entails that publications need to include both of the terms. Using the AND operator increases the likelihood of more publications being reached in comparison to using the terms as a concatenated word sequence.

To capture terms such as "architectural" and/or "architecture", the asterisk character * is used, known as a wildcard, to match one or more inflected forms of the searching term. To increase the likelihood of finding publications addressing architectural aspects of TD, the target of the search term is defined to search in both title and abstract.

The *selection of data sources* involved automatic searching in six well-known digital libraries: the ACM Digital Library, IEEExplore, ScienceDirect, SpringerLink, Scopus, and Web of Science. Additionally, in order to avoid overlooking important publications, we performed a manual hand search in all the proceedings of a key conference on the subject: the International Workshop on Managing Technical Debt (MTD Workshop). The search was conducted in April 2017 and included publication within the timeframe of 2005-2016.

9.3.3. Snowballing

In software engineering, the primary recommended first step is using search strings in a number of databases and thereafter to apply snowballing as a second step [106]. Furthermore, Webster and Watson [107] propose using both a backward and a forward snowballing method, in order not to miss any potentially relevant publications.

Backward snowballing refers to using the reference list of the publications in order to identify additional papers and forward snowballing refers to studies citing the publication [87].

Snowballing is an iterative method where the first iteration uses as input the selected publications from the first filtering stages and examines the referenced papers in subsequent iterations. Each iteration follows the selection criteria in the same way as the study selection phase as described in Section III, based on the title, abstracts, and on full

المنارة للاستشارات

www.r

text. The selected studies from the snowballing process were merged into the final results of the publication selection. The snowballing search was conducted in May 2017.

9.3.4. Inclusion and exclusion criteria

SLRs require explicit inclusion and exclusion criteria to assess the fitness of the content in each possible primary study with respect to the RQs.

The criteria should be based on the research questions, and be applied after the full texts have been retrieved [80]. The inclusion and exclusion criteria that have been used in this study are listed in Table I:

TABLE I - INCLUSION AND EXCLUSION CRITERIA

Criteria	Assessment criteria
Inclusion	Publications should define or discuss architectural issues in the context of TD.
Inclusion	Publications published in journals, in conference proceedings, book chapters, and workshop proceedings were decided to be included.
Inclusion	Only publications written in English were included.
Exclusion	Publications that only mention TD in an introductory statement and do not fully or partly focus on its architectural aspects.
Exclusion	Publication's full text is not available (i.e. if the database used does not allow access to the full text of the publication)
Exclusion	Publications where the full paper is not possible to locate (i.e. if the used database does not have access to the full text of the publication).
Exclusion	For conferences and workshop proceedings, publications earlier than the year 2005, and later than 2016, were excluded.
Exclusion	The publication is a secondary study, i.e. a literature review.

9.3.5. Quality assessment

To ensure that the selected publications comply with a certain quality, all publications went through a quality assessment process to evaluate if they were of an adequate standard. In order to make this assessment, each publication´s content was evaluated using a set of questions in a checklist (Table II), where the answers were mapped according to the options on a ranking scale.

The questions (QA1-3) in the checklist represent different assessment criteria, with a focus on three diverse quality assessments relating to the quality that needs to be

المنارة للاستشارات

www.r

considered when evaluating the publications identified in the review. The assessment criteria strive to appraise the quality of the publication (QA1), quality according to the findings and results (QA2), and relevant to an ATD aspect (QA3).

The scoring procedure for QA1 was Journal = 4, Conference = 3, Book = 2, Workshop = 1, and for QA2-3 it was set at Excellent = 3, Good = 2, Fair = 1 and Poor = 0. The first author coordinated the quality evaluation extraction process, and all authors contributed to the quality assessment. If there were any disagreements during the scoring procedure, the issues were discussed until we reached a mutual agreement.

TABLE II - QUALITY ASSESSMENT CRITERIA

Question	Assessment criteria	Response option scale
QA1	Where was the research published?	Journal = 4 Conference = 3 Book = 2 Workshop =1
QA2	Did the publication provide clearly stated findings with credible results and justified conclusions?	Excellent = 3 Good = 2 Fair = 1 Poor = 0
QA3	Did the publication provide a valuable contribution to the review, in terms of the relevance of discussing ATD?	Excellent = 3 Good = 2 Fair = 1 Poor = 0

The scale of QA1 refers to a well-established standard where the Journals are ranked as having the highest reliability, followed by Conferences, Books, and Workshops. QA3 involves comparable requirements, which were previously referred to as exclusion criteria, but this examination focuses on scaling the content of the remaining publications. The quality assessment scores are heuristic only; to be used as a guide where no publication is rejected on the basis of the quality assessment output.

9.3.6. Data collection and Data synthesis

Data were collected according to the form in Table III, including predefined Data Collection Variables. This enabled the recording and tracking of the full details of each surveyed publication. The predefined Data Collection Variables [DC7] to [DC12] are, as a result, based on a synthesis step in order to refine the classification of the research questions. These variables were iteratively created by examining the full text of the retrieved publications.

المنارة للاستشارات

www.r

TABLE III - DATA COLLECTION VARIABLES AND THEIR PURPOSE

Variable	Collected Information	Purpose
[DC1]	Author	Demographic characterization
[DC2]	Title	
[DC3]	Year	
[DC4]	Venue	
[DC5]	Quality assessment score	Data assessment
[DC6]	Impact of ATD	RQ1
[DC7]	Categories of ATD	RQ2.1
[DC8]	Quality attributes/negative effects	RQ2.2
[DC9]	Refactoring Strategies	RQ2.3
[DC10]	Architectural TDM activities	RQ3.1
[DC11]	Challenges	RQ3.2
[DC12]	Analysis method	RQ3.3

The first data collection variables [DC1] – [DC4] are primarily due to the demographic characterization of the study. Variable [DC5] reveals and synthesizes the quality of the publications, and this score is used when selecting information from the sources as well as when resolving conflicts among contrasting statements in the data. Variable [DC6] reports the impact and importance that ATD has on the software development, in order to address RQ1. The stated ATD categories in the research presented by variable [DC7] provide information when processing RQ2.1; equally, [DC8] reports the quality attributes and possible adverse effects when answering RQ2.2. To scrutinize different refactoring strategies mentioned in the research, the variable [DC9] is added to the data collection. Finally, to answer RQ3 and its sub-queries RQ3.1, RQ3.2 and RQ3.3, variables [DC10], [DC11] and [DC12] with a focus on the different types of the abovementioned ATDM activities together with various challenges and monitoring/detecting methods are provided.

9.3.7. Data analysis

Kitchenham et al. [108] recommend the use of graphical examination of data and exploratory data analysis. For such a purpose, the software tool Atlas.ti is used for qualitative data analysis allowing the coding and visualization of qualitative information. Figure 1 shows the outcome of the analysis process, described below.

المنارة للاستشارات

www.r

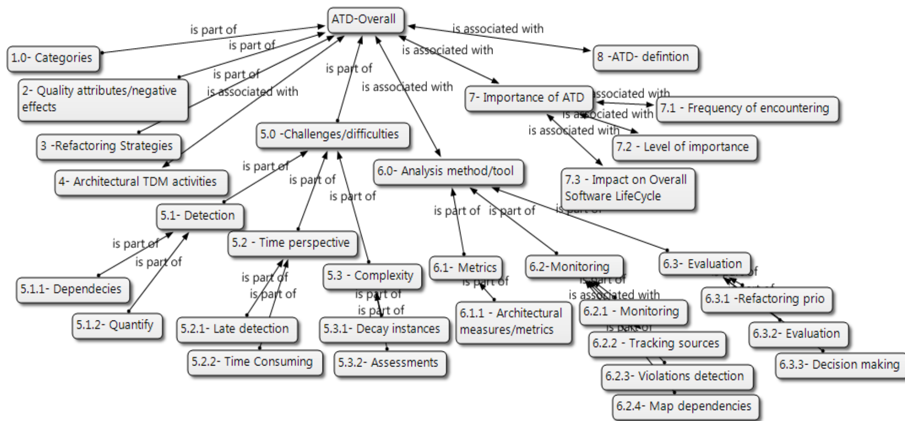


Figure 1. The outcome of the analysis process

First, as reported in the data collection Section (III-F), the RQs generated DC variables (DC6-DC12 in Table III), which were used as an inductive coding scheme for the high-level categorization of the data (second level of codes in Figure 1). Following this, the data were inductively analyzed identifying several aspects, visible as the bottom-level boxes in Figure 1. In order to explain how this critical coding step was conducted, an example of a quotation from a paper mapped to a novel aspect and therefore to an RQ is reported.

The quote from Mo et al. [109] "*Not being able to detect or address architectural decay in time incurs architecture debt that may result in a higher penalty*" was coded as Time Perspective, which is part of the challenges related to ATD management (RQ3.2).

During the analysis, we were explicitly observing cause and effect relations between the revealed aspects, which were introduced as relationships between the aspects. The complete result of this process is the Unified Model in Figure 5, where all the aspects and their relationships are shown in a comprehensive manner.

9.4. Results from the retrieval of publications

The presentation of the results in this section will focus on the overall results concerning the retrieval of the primary publications.

To screen out the most interesting and relevant publications for this review, a filtering technique based on five different stages has been used. Fig. 2 shows the filtering stages included in the searching process and the returned number of publications identified after each filtering stage. To increase the coverage, we searched for relevant publications by conducting both a systematic search of available research publication databases and by using a "snowballing" technique.

المنارة للاستشارات

www.r

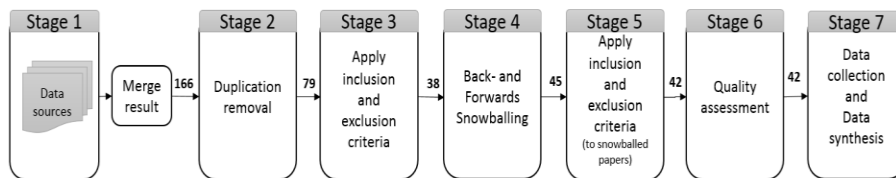


Figure 2. Stages included in the selection process (counts indicate included papers after each stage).

In the first stage, all publications (n = 166) from the different data sources were retrieved and merged. The publications were entered into the EndNote (version 7.7) software tool for managing papers, citations, and references, in a structured manner. Endnote facilitated the second stage of finding and removing duplicates, which resulted in a reduced number of publications (n = 79). Stage three was applied after the full texts were retrieved, and each publication was checked using the inclusion and exclusion criteria in Table I. As a result of this action, another 41 publications were excluded, which returned 38 publications. This set of publications retrieved from research databases was then used as input in stages four and five when applying the snowballing technique.

When applying the backward snowballing method, the reference list in each publication was examined. In total, 812 publications were referenced in the 38 selected publications (although some of the publications were duplicates).

When applying the forward snowballing method, publications citing the 38 selected publications were studied. In total, seven more papers were selected after applying forward- and backward snowballing of the references of the 38 selected publications. From those seven additional publications, three publications were excluded after applying the inclusion and exclusion criteria, which returned 42 publications for a detailed quality assessment.

In the sixth stage, the publications went through an assessment process, with the idea of assessing the quality of all publications. This process did not reduce the number of publications further; this stage predominately serves as a ranking of the publications. The overall output of the first six stages returned 42 publications for data analysis. During the seventh stage, data were extracted from each of the 42 primary publications included in this SLR, according to the predefined data collection forms, in Table III.

According to Kitchenham [80], a synthesis of the SLR procedure should include a descriptive synthesis and also possibly a quantitative summary (meta-analysis). Below, Table IV, Figure 3 and Figure 4 present the extracted information and statistics regarding the retrieved publications.

The selected venues and publication types are listed in Table IV. The examination of the selected primary publications showed that the most predominant type is conference papers (45%), followed by workshop papers (36%), and six (14%) publications were published in journals. Two chapters from two books were included.

المنارة للاستشارات

www.r

TABLE IV - SELECTED VENUES

Venue	ID	Publication	Type
Journal: Information and Software Technology	INFSOF	[110]	Journal
Journal: IEEE Software	IEEE	[111],[112]	Journal
Journal: IEEE Software	IEEE	[18]	Journal
Journal: The Journal of Systems and Software	JSS	[113]	Journal
Journal: Management Science	MS	[114]	Journal
International Conference on Dependable Systems and Networks	DSN	[115]	Conf.
Joint Meeting on Foundations of Software Engineering	ESEC/FSE	[16]	Conf.
Proceedings of the Annual Hawaii International Conference on System Sciences	HICSS	[24]	Conf.
International Conference on Software Engineering	ICSE	[116]	Conf.
International Conference on Software Engineering	ICSE	[54],[117]	Conf.
International Conference on Software Maintenance	ICSM	[118]	Conf.
International Conference on Quality of Software Architectures	QoSA	[119]	Conf.
International Conference on Quality of Software Architectures	QoSA	[26]	Conf.
International Conference on Quality of Software Architectures	QoSA	[120]	Conf.
International Conference on Quality Software	QSIC	[28]	Conf.

المنارة للاستشارات

www.r

Venue	ID	Publication	Type
Euromicro Conference Series on Software Engineering and Advanced Applications	SEAA	[93]	Conf.
Software Engineering and Advanced Applications, Euromicro Conference	SEAA	[121]	Conf.
Software Engineering and Advanced Applications, Euromicro Conference	SEAA	[122]	Conf.
Conference on Software Architecture and European Conference on Software Architecture	WICSA-ECSA	[97]	Conf.
International Conference on Software Architecture	WICSA	[103],[23],[123]	Conf.
International Conference on Systems, Man, and Cybernetics	SMC	[124]	Conf.
International Workshop on Managing Technical Debt	MTD	[125]	Ws.
International Workshop on Managing Technical Debt	MTD	[126],	Ws.
International Workshop on Managing Technical Debt	MTD	[25]	Ws.
International Workshop on Managing Technical Debt	MTD	[109]	Ws.
International Workshop on Managing Technical Debt	MTD	[17]	Ws.
International Workshop on Managing Technical Debt	MTD	[127]	Ws.
International Workshop on Managing Technical Debt	MTD	[128],[129],[130]	Ws.
International Workshop on Software Architecture and Metrics	SAM	[131],[132],[131], [27]	Ws.
International Workshop on Technical Debt Analytics	TDA	[133]	Ws.
Economics-Driven Software Architecture		[22]	Book

المنارة للاستشارات

www.r

Venue	ID	Publication	Type
Software Quality Assurance		[96]	Book

Fig. 3 shows the number of publications per venue included in the review. The graph reveals that the most prominent venue is the Managing Technical Debt Workshop (MTD, with nine publications) followed by the Conference on Software Architecture and European Conference on Software Architecture (WICSA/ECSA), with four publications.

Fig. 4 shows the distribution of the publications from 2005 to 2016. This figure shows an apparent increasing trend for publications in the field. For instance, between 2014 and 2015, the amount of publications more than doubled.

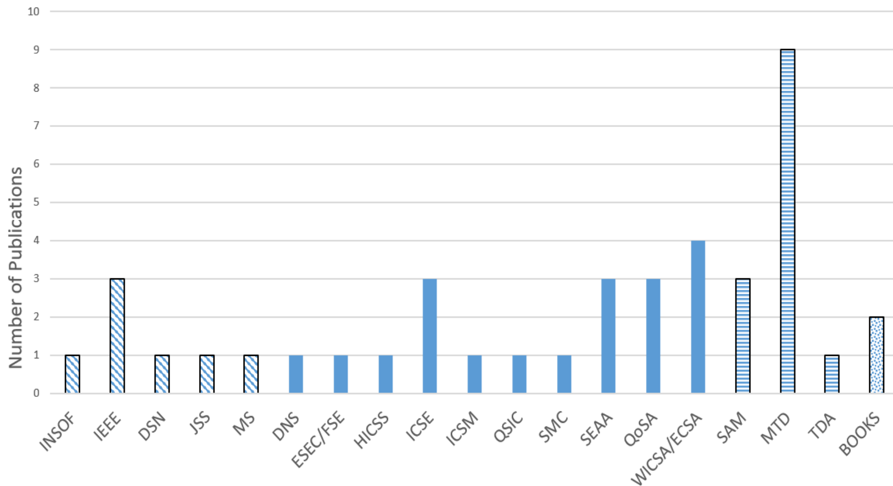


Figure 3. Numbers of publications per venue

المنارة للاستشارات
www.r

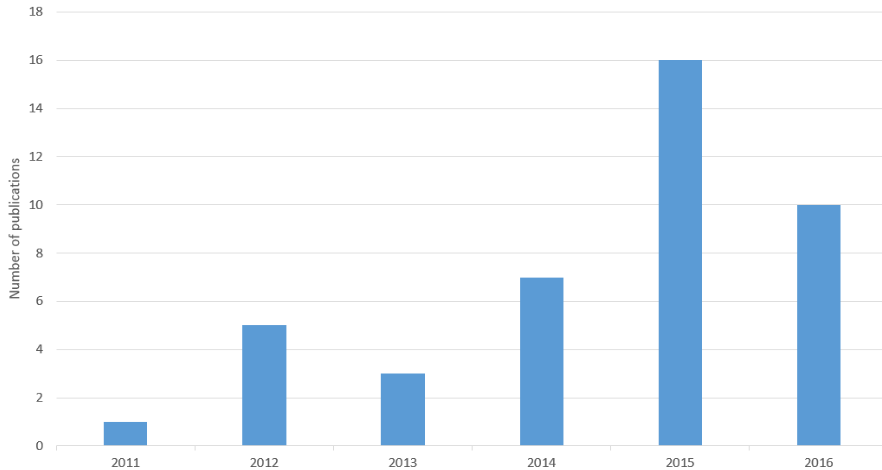


Figure 4. Publications per year

9.5. Results

In this section, we present the results of the data analysis of the reviewed publications, with respect to the investigated research questions RQ1-3. This section provides a more detailed analysis and review of the selected 43 publications.

9.5.1. What is the importance of ATD in software development? (RQ1)

This research question seeks to address the relevance and importance of ATD in software development. These days, ATD is considered to be central to software development, and ATD has a significant negative impact on software products and on software development projects. Left unchecked, ATD can cause expensive repercussions. It is therefore important to understand the basic underlying factors of ATD.

ATD is an important component that needs to be addressed in the architecting process [96], [117],[133],[131],[16], [128],[28],[22],[113],[119] in order to make sure that the advantages of sub-optimal solutions do not lead to the payment of significant interest [127]. [131] highlights that architectural decisions concerning different conflicting quality attributes often become major sources of significant TD and are an expensive form of architectural breakage to rectify. Furthermore, Izurieta et al. [120] describe that, for practitioners adopting model-driven techniques, the management of TD also requires addressing this problem during the specification and architectural phases. In our previous paper [93], we pointed out the risks when ATD grows until the negative effect makes adding new business value so slow that it becomes necessary to conduct widespread refactoring or even rebuilding the software from scratch.

Several of the reviewed publications highlight the importance of addressing the architectural aspects of TD since it has such leverage within the overall development lifecycle and its strategic management [28], [109]. Furthermore, the reviewed publications state that the most commonly encountered instances of TD are caused by

المنارة للاستشارات

www.r

architectural inadequacies [17] and that architectural decisions are the most important source of TD [16]. [18] echo the notion, stating, "*Architecture plays a significant role in the development of large systems, together with other development activities, such as documentation and testing (which are often lacking). These activities can add significantly to the debt and thus are part of the technical debt landscape*".

9.5.2. Existing knowledge about ATD regarding debt, interest, and principal (RQ2)

RQ2 investigates how existing knowledge defines debt, interest, and principal, which are respectively represented by the three following aspects: (1) categories of ATD, (2) the negative effects caused by ATD, and (3) strategies for refactoring ATD.

9.5.2.1. Categories of ATD (RQ 2.1)

Architectural choices and design decisions have a great impact on the amount of ATD [134], [115], and can be incurred by either explicit or implicit architecture decisions [22] and be made either consciously or unconsciously [26]. These decisions affect several different categories of debt, and one of the main categories of ATD is architectural dependencies [23], [126] including module dependencies, external dependencies, and external team dependencies [16]. Another category of ATD involves non-uniformity of patterns and policies where, for example, a violation of naming conventions and non-uniform design or architectural patterns are implemented [23]. Some authors add code-related issues as ATD variables, where code duplication [23] and overly complex code [25],[132],[118] are additional reasons for the emergence of ATD. Furthermore, non-uniform management of integration with subsystems and resources [23] and different architectural decay instances [109] are also revealed as ATD categories. The lack of implementation or test of Quality Attributes (QA) or non-functional requirements are shown by [23] as a classification of ATD, and [131] illustrates the problems with conflicting QA synergies as an important source of ATD.

9.5.2.2. Negative effects caused by ATD (RQ 2.2)

Fernández-Sánchez et al. [24] acknowledge that "*interest is the recurring cost of not eliminating a technical debt item over some period of time*" with the result of the negative effects and adverse impact on QAs. There is wide agreement in academic literature that some of the negative consequences of ATD can be linked to the effect it has on maintenance complications and penalties, and in stifling the organization's ability to introduce new features: "*Not being able to detect or address architectural decay in time incurs architecture debt that may result in a higher penalty in terms of quality and maintainability (interest) over time*" [109]. Li et al., [103] mention, in particular, the QA's maintainability and evolvability, as immature architecture design artifacts. Similar results from research by Xiao et al. [117] show that a significant portion of the overall maintenance effort is consumed by paying interest on ATD. Moreover, Avgeriou et al.

المنارة للاستشارات

www.m

[111] draw on the work of Martini et al. [110] who identify a relationship between architectural shortcuts and potentially higher evolution and maintenance costs.

A system suffering from architectural drift or erosion will eventually develop some decay instances that negatively impact the system's lifecycle properties, such as understandability, testability, extensibility, reusability [109] and reliability [114].

[112] reveals that architectural compromises often occur when architectural constructs are implemented too late, causing the solution to accumulate TD and making it increasingly difficult to add features in an evolutionary manner. Naturally, the time dimension is crucial to achieving the right amount of anticipation.

On the other hand, to proactively assume that changes will happen and design for flexibility in advance often entails high costs and risk [24]. Although the large majority of the surveyed publications only highlight the negative consequences of ATD, we find four publications explicitly mention a positive impact in terms of strategic benefits to an enterprise when deliberately taking on ATD, in terms of shorter time to market [22], [125], [18], [111].

9.5.2.3. Refactoring strategies for managing ATD (RQ 2.3)

Decisions of if and when to refactor architecture are extremely important and difficult to take, since changing software at the architectural level is somewhat expensive [54]. The concept of different refactoring strategies refers to how (i.e. if one refactoring process must be performed before another [119]) and whether to repay the debt and finally how to identify candidates for refactoring.

A refactoring strategy typically involves decisions regarding continuing paying interest or by paying the principal by re-architecting and refactoring to reduce future interest payments [17],[122]. In our previous paper, by Martini and Bosch [110], a description of a strategy with respect to minimizing risk for a development crisis is recommended: "*Partial refactoring is the best option for the maximization of refactoring. Thorough refactoring is not realistic. Drastic minimization of refactorings (No refactoring) often leads to development crises in the long run."* Based on economic considerations, it could be more profitable to delay the refactoring, i.e. continue paying interest, than to invest in refactoring to manage the debt and, furthermore, the team's capacity to perform the refactoring should be considered [24]. Several authors mention this decision-making as the main refactoring strategy and [121] reveal important aspects during this prioritization regarding lead time, maintenance costs, and risks. The refactoring decision-making is enclosed by the problem that costs are concrete and immediate whereas the benefits of refactoring are vague and long-term, and the benefits of refactoring have historically been difficult for architects to quantify or justify [116]. [113] state that "*While a software architect might intuitively recognize the potential benefits of architectural change, senior managers typically require a robust assessment of the financial consequences of change, before funding such efforts*." Furthermore, [113] address the questions if and when it might be appropriate to perform refactoring. They highlight the lack of robust data by which to evaluate the relationship between architectural design choices and system maintenance costs, to predict the benefits that might be released through refactoring

المنارة للاستشارات

www.r

efforts. By analyzing the relationship between system architecture and maintenance costs, they show that measures of coupling are a strong predictor of subsequent file maintenance.

Martini et al. [122] describe that, even if the achieving modularity is an expensive operation, the cost of modularization by refactoring of the software architecture could be repaid in several months of development and maintenance. When identifying candidates for the architectural refactoring in order to minimize the refactoring effort, Choudhary and Singh [133] use a refactoring strategy focusing on locating the architecturally relevant classes as they are considered to be the pillar classes of the software design. Their strategy is based on a combination of finding classes that have earlier been frequently refactored together with looking for classes which are harmful to the system's architectural design. The classes are then prioritized and ordered according to their impact on the overall system's quality. When prioritizing among different possible architectural refactoring issues, Martini et al., [54] address the importance of also taking into consideration the available resources in relation to feature development by addressing if and when refactoring should be performed. Martini and Bosch [54], citing Carriere et al. [53] conclude that doing architectural refactoring is risky, difficult to estimate, and difficult to prioritize.

9.5.3. Existing challenges and solutions in managing ATD (RQ3)

RQ3 is separated into three top-level precedents, examining ATDM activities, related challenges, and analysis methods for detecting or evaluating ATD.

9.5.3.1. Architectural TDM activities (RQ 3.1)

Nord et al. [97] describe the importance of this process as "*Development decisions, especially architectural ones, require active management and continuous quantitative analysis, as they incur implementation and rework cost to produce value*". To fully manage and control a complete ATDM process, Li et al. [22] advocate five different activities: identification, measurement, prioritization, repayment, and monitoring. However, while the concept of ATDM as an overall process is uncommonly described in the academic literature, several authors mention or focus on individual activities within an overarching process of ATDM.

The initial activity within the ATDM process is to identify current ATD items within the software system [22], where this activity is crucial for a forthcoming successful management process. The following ATD measurement activity examines and estimates the costs and benefits, including the prediction of change scenarios influencing ATD items for interest [22]. The output of this activity is used as an input to the prioritization activity, since prioritizing refactoring ATD items is an essential element when balancing the short-term value delivery and the long-term responsiveness where lead time, maintenance costs and risk are the variables that most influence the ATD effects[135]Eliasson et al. [127] describe how graphically visualizing the prioritization and evaluation of ATD could increase the awareness of the impact that ATD had on efficiency. The repayment activities involve refactoring and can either be partially or fully

المنارة للاستشارات

www.r

resolved by making new, or modifying existing, architecture decisions [22] to reduce or mitigate the undesirable effects of ATD. The monitoring activity tracks ATD changes explicitly and consequently keeps all the ATD items of the system under control [22]. This activity is a complex process that endures over time, and [97] enforces that the repayment and the monitoring activities are a quite uncommon practice where existing techniques are lacking.

9.5.3.2. Challenges with ATD (RQ 3.2)

To obtain a rich picture and to reveal issues and risks early [27], it is of vital importance to understand the background of ATD problems. This highlights the need to understand the challenges in fully managing ATD in a more conscious and informed way. Mo et al. [109], among others, discuss the challenges and put the issue of time into perspective in ATD. Ernst et al. [16] explain that architectural decisions take many years to evolve and are commonly made early in the software lifecycle and it is often invisible until very late in the lifecycle [28]. Yet another time-related perspective to ATD is offered by [124]. They have studied software installed in automated production systems and describe the challenges with software that is installed at a customer site for the first time, which leads to decisions under time pressure since these software projects commonly face heavy penalties in case of failure. This time pressure could lead to neglecting architectural concepts. Xiao et al. [117] advocate that projects could save a significant amount of maintenance costs if they can discover ATD early, and pay them down by refactoring.

From a more technical perspective, critical issues relating to challenges of the detection of architectural issues through standard testing [25], [18] and ATD seldom yield observable behaviors to end users [26], whereas a higher number of dependencies decreases the comprehensibility of the system [120]. Eliasson et al. [127] describe that inconsistency between different levels of abstraction in the architectural design is difficult to detect and is an important source of ATD. They also highlight that the interest is hidden from the stakeholders, which make it difficult when deciding if a refactoring of an architectural issue should, or should not, be refactored.

Yet another purported emergence of ATD is associated with architectural decision-making, where the decisions cross every possible communication link across the development and operations network, and loss of essential information is practically inevitable [123].

9.5.3.3. Analysis method for detecting or evaluating (RQ 3.3)

Using an analysis method may facilitate the evaluation and decision-making process, helping in the prioritization of resources and efforts concerning refactoring strategies. The need for supporting tools for system monitoring and evaluating ATD using accurate metrics is a key issue and is not fully supported by any currently available tools [16], [23], [130],[129], [111].

The main goal of a continuous and iterative system monitoring of ATD is to capture and track the presence of ATD within a system [23], to provide early warnings to detect costs

المنارة للاستشارات

www.m

and risks [97] and to map architectural dependencies or pattern drift to decay [16]. Moreover, Ozkaya et al. [27] state that most available metrics are at the code level and Reimanis et al., [130] conclude that currently available TD tools only focus on the structural aspects of the system instead of also including the behavioral aspects, while Nord et al. [97] describe that the existing metrics for system quality visibility are insufficient and unproven.

Fontana et al. [128] have analyzed different available TD Index tools to understand which tools take the architectural issues more into account. They found that even if there are some available tools addressing architecture-related issues, most of them do not provide indexes that are directly useful when evaluating single projects. Fontana et al. conclude that the outcome measures from the tools could not be interpreted with the aim of understanding the overall quality of the analyzed project on a global scale [128].

9.6. Importance of ATD and a need for a Unified Model

There is wide agreement in the reviewed academic literature that ATD is of primary importance. However, it is observed from the result that ATD is described in a scattered and inconsistent way. Consequently, we conclude that to derive more value from the results concerning ATD and its effects, a holistic model depicting different views and their implications at hand is required. Therefore, we propose a novel descriptive model that provides an overall understanding of existing knowledge in the research area of ATD with the aim of providing a comprehensive interpretation of the ATD phenomenon.

To interpret the SLR outcomes for practice, and to be of use to practitioners, we have developed a model for translating the results to an organization's needs for understanding ATD [136]. The model in Fig. 5 graphically illustrates and synthesizes the findings and their causal relationships. The structure of the model is inspired by Nickerson et al. [137] who recommended that a taxonomy of a model includes the criteria of conciseness, inclusiveness, comprehensiveness, and extendibility, Here, the conciseness refers to the fact that our model contains a limited number of dimensions and characteristics in order to be easy to comprehend and apply. The inclusiveness refers to the fact that the model includes a sufficient amount of information to illustrate a summarizing view of the research state of the art on the architectural aspect of TD from a holistic perspective. The comprehensiveness refers to the fact that the model provides an overview of all identified aspects. Finally, the model is extendable in terms of the fact that the model allows for additional aspects and relationships to be added.

The model also clarifies the different aspects of each research question and assembles relationships between them. As an aid to obtaining levels of observation, the model is divided into the vertical groups of *ATD Identification Checklist*, *ATD Impediments*, and *ATD Management*, and the horizontal cross-sections corresponding to *Focus area*, *Research question*, and *Aspects*. Each aspect of the model is derived from results obtained from research questions RQ2 and RQ3 and is illustrated in the horizontal cross-sections named RQ.

Within every aspect in Fig.5, there is a box with a number indicating how many papers fit into each aspect. This information highlights the popularity of each categorization and

المنارة للاستشارات
www.m

contributes to the reader's knowledge base with useful and important information when creating a platform for understanding ATD. The references for each aspect are presented in Table V and also reported in the results, Section IV.

The model's scaffolding allows practitioners to more easily leverage this study's outcome when analyzing the results derived from the RQs. The model's relationships are based on the systematic analysis of all surveyed publications, in which we have interpreted the result as the unified model illustrated by different types of arrow. Only the relationships that are clearly stated and described within the retrieved publication are illustrated in the model. This implies that we cannot exclude that there are other relationships between the aspects. The identified relationships between the aspects are illustrated by different colored arrows. The prospect of being able to identify more relationships among the different aspects serves as a continuous incentive for future research. The model highlights that Maintainance and Evolvability are vital challenges within ATD, due to the fact that all of the ATD challenges (green dotted arrows) are related to this negative effect, and furthermore, all of the ATD categories have an effect on Complexity (orange dashed arrows).

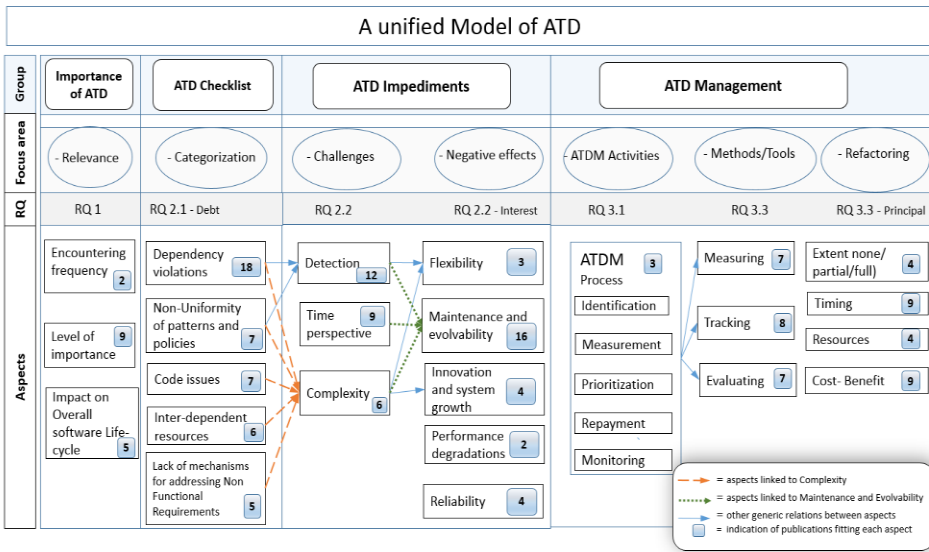


Figure 5. The Unified Model of ATD

9.7. Discussion

This section discusses the findings and the implications for practitioners and academia. The results from the SLR indicate that there is an absence of a comprehensive descriptive model of ATD in the academic literature. For the academic and practitioner community, this descriptive model can support the process of more informed management of the software development lifecycle, with the goal of raising the system's success rate and lowering the rate of negative consequences.

المنارة للاستشارات

www.m

The remainder of this section presents and discusses the answers to the research questions presented in Section III-A, using the grouping described in Fig. 5 and, as the final section, the novel Unified Model is discussed.

TABLE V - REFERENCES FOR EACH ASPECT

Aspect	Reference
Frequency of encounterings	[17], [117]
Level of importance	[131],[16],[128],[28],[18],[22],[96],[113],[119]
Impact on Overall software Lifecycle	[133],[127],[120],[93],[109]
Dependency violations	[115],[126],[127],[16],[120],[116],[103],[96],[26],[113], [23], [121],[93],[110],[97],[109],[123],[117]
Non-Uniformity of patterns and policies	[24],[132],[120],[22],[26],[93],[124]
Code issue	[111],[25],[132],[23],[121],[97],[124]
Inter-dependent resources	[131],[23],[110],[114],[109],[124]
Lack of mechanisms for addressing Non-Functional Requirements	[131],[127],[96],[23],[124]
Detection	[111],[126],[127],[132],[18],[103],[96],[26],[118],[109], [124],[117]
Time perspective	[103],[96],[23],[93],[110],[97],[112],[124],[117]
Complexity	[115],[126],[132],[28],[26],[27]
Flexibility	[131],[24],[109]
Maintenance and evolvability	[111], [127],[132],[18],[22],[103],[96],[26],[118], [113],[23],[93],[110],[112],[117]
Innovation and system growth	[18],[113],[93],[110]
Performance degradations	[25],[127]
Reliability	[131],[25],[114],[109]
ATDM Process	[22],[103],[96]
Measuring	[16],[132],[120],[116],[22],[96],[110],[130],[117]
Tracking	[16],[116],[22],[103],[96],[23],[110],[117]
Evaluating	[127],[16],[128],[116],[22],[110],[130]

المنارة للاستشارات

www.r

Extent none/partial/full)	[113],[23],[93],[110]
Timing	[132],[23],[54],[121],[93],[110],[122],[97],[117]
Resources	[116],[54],[110],[124],
Cost-Benefit analysis	[116],[113],[54],[121],[93],[110],[122],[97],[119]

9.7.1. Importance of ATD (RQ1)

The first research question set out to review how the importance of ATD is described in the retrieved publications. The results of this study show that ATD is of very high importance to software companies and that ATD has a negative impact during the overall software lifecycle. Some publications also stated that, among the different types of TD, ATD is the most commonly encountered type of TD. Since ATD has such leverage within software development and that it is of primary importance to pay attention to it, we conclude that ATD needs further attention from both academia and industry.

9.7.2. ATD identification checklist (RQ2.1)

The research question RQ2.1 is addressed with an ATD identification checklist, in the form of a unified model (Fig.5). The current description in the literature of ATD is inconsistent and creates confusion both for practitioners and within academia. Therefore, we provide a comprehensive checklist of all aspects of ATD, reported in the literature. Researchers and practitioners can use this checklist as a universal reference for recognizing ATD. ATD can be categorized into different categories of debt (RQ2.1), such as: dependency violation, code related issues, non-uniform usage of architectural policies, the lack of handling interdependent resources, and lack of addressing non-functional requirements.

9.7.3. ATD impediments (RQ3.2 and RQ2.2)

In Fig. 5, we present all the ATD Impediments – both the challenges related to ATD and their associated negative effects. Researchers and practitioners can use this picture to evaluate and understand what problems might occur when dealing with ATD and the consequences of these challenges are left unattended. Major challenges (RQ3.2) include difficulties in ATD detection, challenges related to the time perspective, and unmanageable architectural complexity. Frequently, the mentioned complexity is related to code level (McCabe's cyclomatic complexity) but this metric is not related to architectural aspects of complexity, and, therefore, we stress the need for further focus on this specific aspect.

These overall challenges can lead to severe and costly maintenance overheads that, in the long run, can diminish the organization's ability to innovate and evolve. ATD can also result in restricted flexibility, decreased reliability and performance degradation (RQ2.2). Our findings show that challenges and effects are currently presented in a scattered way

المنارة للاستشارات

www.r

in the literature, and, therefore, we provide a comprehensive presentation of ATD-related challenges and effects in Fig. 5.

9.7.4. ATD management (RQ3.1, RQ3.3, and RQ2.3)

Although the current literature stresses the importance of understanding ATD and its related consequences, we report a lack of guidelines on how to manage ATD successfully in practice.

We confirm that the generic activities recognized by Li et al. [22] apply to ATD management. However, we report a lack of an overall process where these activities are fully integrated. At the moment, they are generally reported as single isolated activities. Two book chapters mention these activities within an overall process, but there is a need for further and stronger empirical evidence supporting a suitable and practical solution (RQ3.1).

The findings show that there is a compelling need for supporting tools and methods for system monitoring and evaluating ATD, but also show that no software tools covering the full spectrum of ATD are yet available (RQ3.3). For example, Li et al. [22] propose an ATD identification method, but it is not clear to us how this method could be integrated with other methods and tools.

A refactoring strategy for paying the principal must be developed and implemented to successfully manage ATD and not allow it to grow unimpeded. Practitioners do, in general, lack strategies for architectural refactoring, and, therefore, such an activity might result in an ad-hoc process where the results are inadequate. In this paper, we provide the key dimensions that need to be taken into consideration when defining such a refactoring strategy. Fig. 5 includes these dimensions and can assist practitioners and researchers when creating and studying refactoring strategies. A strategy needs to be flexible to adapt to changing requirements and take into account resource constraints and forthcoming new features. An important aspect to consider while setting up a strategy is the amount of refactoring that should take place. If refactoring is overlooked, it can lead to a development crisis in the long run, and there are benefits of performing a partial refactoring (RQ2.3).

9.7.5. The Unified model for ATD

As discussed in the previous section, we found that the information about ATD is currently dispersed across different publications and is sometimes inconsistent. It is therefore difficult for researchers and practitioners to obtain a clear overview of ATD.

This study provides a novel and comprehensive model that includes all the important aspects of the ATD phenomenon and their relationships (Fig. 5). The model will help academics and practitioners in interpreting ATD, recognizing its issues, and understanding how to manage it. Having such a visual model in which stakeholders can rapidly obtain a holistic overview is of immense value.

The model presents different aspects of ATD and their relationships: this helps the practitioners in understanding how the aspects impact each other and assists the

المنارة للاستشارات
www.r

researchers in studying the connecting aspects together. For example, the orange dashed arrows in the model show that all revealed categories of ATD have a relation to the aspect of Complexity, meaning that all instances of ATD increase the needed effort for software and system engineers to understand and manage the systems' interfaces and interconnections.

The green dotted arrows in the model show that all the Challenges of ATD have effects on system Maintenance and Evolvability, meaning that every ATD item will have a negative impact on Maintenance and Evolvability.

9.7.6. Research agenda for ATD

This research agenda provides clear targets for future research within the ATD field and concludes that future research roadmaps should focus on:

- The indication of how many papers fit each aspect in the unified model of ATD clearly shows which areas are covered by research and what aspects need more investigation.
- The findings show that there is a compelling need for supporting tools and methods for system monitoring and evaluating ATD. There is also a need for more studies concerning efficient refactoring strategies, taking different aspects into consideration.
- The results of this study also underline the lack of guidelines on how to manage ATD successfully in practice, which needs to be addressed in future research.

Very few publications stress the importance that organizations need to deliberate reverse TD and remediate their software solutions to avoid facing expensive repercussions in the future.

9.7.7. Threats to validity

The result of this SLR may be affected by some threats to validity, such as:

9.7.7.1. Incompleteness of study search and obtaining accurate data bias

In a literature review, it is important to mitigate a potential bias of the poor quality of publications, which can lead to inaccurate conclusions [138]. Therefore, the included publications were published in journals, conference proceedings, or workshop proceedings according to a peer review process. Some book chapters were also included in order to include all relevant publications about ATDM.

In order to mitigate the risk of not retrieving relevant publications, which could negatively affect the completeness of the study, we searched the most common electronic databases in which a large number of journals and conference and workshop proceedings, along with book chapters in the software engineering field, are indexed. In addition to this, we employed both a backward and a forward snowballing technique to include additional

المنارة للاستشارات

www.r

potential studies in the references of the selected studies retrieved from the database searches.

9.7.7.2. Number of retrieved publications

A relatively limited number of publications were retrieved, with the logical consequence that the results of this review had limited publications from which to derive results. This result could potentially have introduced subjective conclusions into the analysis or incorrect or missing relationships into the results.

9.7.7.3. Search string

During the search process, publications that did not include either of the search terms "technical debt" and architec* in the title or abstract of the publication were excluded from the review. This could imply that some publications could have been incorrectly excluded. We reason, nevertheless, that if a publication primarily focuses on the architectural aspects of TD, the word architec* should be explicitly mentioned. However, we are aware of the fact that there are related terms that, in many respects, resemble the architectural aspects of TD.

9.7.7.4. Data extraction

To reduce the risk of subjectivity during the classification and extraction phase, performed by only one researcher, several publications were examined by at least two researchers to ensure that the returned publications were suitable and equivalently analyzed.

9.7.7.5. The unified model of ATD

In the Unified Model of ATD in Fig.5, other aspects, relationships, and associated impacts may exist but are omitted in this model, since they were not explicitly revealed in the review process and thus have not been further analyzed.

9.8. Related work

Architecture plays a significant role in the development of large software systems yet, to the best of our knowledge, despite the fact that ATD is commonly recognized as a major dimension of TD [7], and except for our previous paper [101], there are no other literature reviews summarizing the research state of the art with a specific focus on the *architectural* aspect of TD, from a holistic perspective.

By studying the types of available literature review, six reviews (written in English) were found within the shared research area of this study. These studies differ, however, in the research questions, method, searching interval, and primary focus, and the studies have different research goals. The most recent study was a systematic mapping review performed by [138] focusing on the elements required to manage TD. This paper is an

المنارة للاستشارات

www.r

extension of [139], and the paper reviews techniques and methods for TD management from an architectural perspective, using a systematic mapping method. Their study addresses code debt, environmental debt, knowledge distribution and documentation debt, and testing debt and with a specific focus on design and architectural debt. The different TD types are studied from three different perspectives: core elements, implementation elements, and management elements. Their analysis shows that further studies addressing architectural debt using a holistic approach are needed. The perspective in our study is different from that study, since we are specifically interested in the architectural aspects of TD and the activities used in performing architectural TD management.

Another study was a systematic mapping review performed by Alves et al. [42], focusing, among other aspects, on different TD types and useful indicators for detecting TD. This study focuses on TD in general, in comparison to our study, which focuses on the architectural aspects of TD. Another significant related work in this research area is conducted by Ampatzoglou et al. [30] based on a comprehensive, systematic literature review with a primary focus on the financial aspects of managing TD. This review also comprises a formation of a glossary of terms and a classification of financial approaches in TD management. Li et al. [15] recently published a mapping study (MS) with the primary focus on making a classification and thematic analysis on collected studies within the TD management area. This study focuses on managing activities, approaches, and tools on TD in general and not especially on the architectural aspects of it. In comparison to our study, this study does not include any ATD impediments in terms of challenges or negative effects. Finally, in 2013, Tom et al. [7] published a multivocal literature review (MLR) with a focus on the nature of TD and its implications for software development. This study focuses on TD in general in terms of different dimensions of TD, how TD arises, and the benefits and drawbacks of allowing TD to accrue.

However, even if one of the most commonly encountered instances of TD is caused by architectural inadequacies, no previous study has specifically addressed this aspect of TD using a holistic approach focusing on categorizations, impediments, management activities, methods, and refactoring strategies.

Even if some of the abovementioned publications refer to the architectural aspects of TD, none of them have a specific and dedicated focus on solely the architectural aspects of TD and thus our study is different from these previous studies, since we are specifically interested in the *architectural* aspect of TD. Our study also provides a unified model which offers a dedicated focus on ATD. The aim of this study is, therefore, to synthesize and compile research efforts with the goal of creating new knowledge with a specific interest in the ATD field, which contributes to both academia and practitioners in terms of the unified model of ATD.

9.9. Conclusions

From a software lifecycle perspective, it is of vital importance to understand and proactively manage ATD. Left unchecked, ATD can potentially stifle the implementation of new features, and organizations may face expensive repercussions due to costly

المنارة للاستشارات

www.r

architectural maintenance efforts. In order to synthesize and compile the current 'state of the art' in the ATD field, we have conducted a systematic literature review focusing on the research areas: ATD in terms of principal, interest, debt and related challenges and solutions for managing ATD.

In this study, we have provided a new and comprehensive understanding and raised awareness regarding the challenges that surround ATD and, finally, how ATD can be successfully managed. The findings showed that there is a wide agreement in the reviewed literature that ATD is of primary importance. ATD is, however, surrounded by several challenges, and while numerous publications mention different isolated ATD management activities, there is an absence of, and a need for, a thorough indicative ATDM process for the practitioner and academic communities, covering all these separate activities. Different ATD categories (as debt) can result in various negative consequences (as interest), requesting effective refactoring strategies (as principal). A refactoring strategy mainly refers to *how* to, and *if*, and to what extent, repaying the debt should be formulated.

This research contributes to the knowledge that addresses a current gap in understanding, where knowledge and research of ATD are non-unified and fragmented. One key contribution of this paper is our novel model of ATD. This model summarizes our findings and allows for the improved identification of ATD and associated negative consequences and corresponding ATDM activities. The model illustrates ATD in a unified and comprehensive way, by exploring different aspects and relationships which are considered particularly valuable for managing and raising awareness about ATD.

The model reveals that all categories of ATD (as debt) are related to the challenge of complexity and furthermore that all challenges are related to maintenance and evolvability. This model can help several different stakeholders within the software lifecycle process to be better and more informed in managing the software, with the goal of raising the system's success rate and lower the rate of negative consequences.

As future work, we plan to investigate this area further by means of expanding this review to include additional closely related research publications. Moreover, we will continue our research in the direction presented in our research agenda.

To further study the impact and the influence of the different aspects of the unified model of ATD, we are currently conducting empirical research with the aim of finding which aspects are the most hurtful during the software development lifecycle.

المنارة للاستشارات

www.r

10. Technical Debt from a Software Quality Perspective

The aim of this chapter is threefold: to understand which quality issues have the most negative impact on the software development lifecycle process, to determine the association of these quality issues in relation to the age of the software, and relate each of these quality issues to the impact of different TD types.

Software companies need to produce high-quality software and support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered by compromised software quality attributes that have an important influence on the overall software development lifecycle. This paper reports the results of six initial group interviews within total 43 practitioners, an online web-survey provided quantitative data from 258 participants and seven follow-up group interviews within total 32 industrial software practitioners. First, this study shows that practitioners identified maintenance difficulties, a limited ability to add new features, restricted reusability, and poor reliability, and performance degradation issues as the quality issues having the most negative effect on the software development lifecycle. Secondly, we found no evidence for the generally held view that the Technical Debt increases with age of the software. Thirdly, we show that Technical Debt affects not only productivity but also several other quality attributes of the system.

This chapter has been published as:
Time to Pay Up - Technical Debt from a Software Quality Perspective
Terese Besker, Antonio Martini, and Jan Bosch
In Proceedings of the 20th Ibero American Conference on Software Engineering (CibSE) @ ICSE17, 2017.

10.1. Introduction

Technical debt (TD) is recognized as a critical issue in today's software development industry and is evidently detrimental to the software developing companies, and it is, therefore, important to assess and estimate the negative consequences of Technical Debt in terms of continuously paying interest.

Ward Cunningham [91] introduced the financial metaphor of Technical Debt (TD) to describe to nontechnical product stakeholders the need for recognizing the potential long-term and far-reaching negative effects of the immature code that is made during the software lifecycle, where the debt has to be repaid with interests in the long term.

Interest refers to the negative effects of the extra effort that has to be paid due to the accumulated amount of TD in the system, such as executing manual processes that potentially could be automated or spending excessive effort on modifying unnecessarily complex code, performance problems due to lower resource usage by inefficient code, and similar costs [7].

المنارة للاستشارات

www.r

Left unchecked, TD can result in unexpectedly large cost overruns, severe quality issues, inability to add new features [50] and even result in reaching a crisis point when a huge, costly refactoring or the replacement of the entire system needs to be undertaken [5]. The negative effects of continuously paying interest during the software lifecycle are burdensome and costly for software companies. Therefore, it is of vital importance to understand and estimate the interest in order to proactively manage it in terms of allocating time, resources and additional effort.

A basic understanding of the structure and the constituent elements of the interest can be obtained by studying the interest from the perspective of "*what kind of interest is to be paid*", "*which type of Technical Debt cause the interest*," and "*when will the interest being paid*."

Today, there exists little quantitative data and knowledge with a focus on the interest payment of TD, in terms of compromised software quality attributes. To the best of our knowledge, there is no research studying the interest from a quality perspective, focusing on the software quality issues, causes of the interest, and the interest in relation to the age of the software.

We, therefore, aim at answering the research questions (the results from RQ1, are used as input in the next following research questions):

- **RQ1**. Which quality attributes are the most affected by TD and how frequent are those encountered during the software lifecycle and are those issues explicitly address within software companies?
- **RQ2**. What type of TD generates the most negative impact on the daily software development work and what kind of interest is related to each TD type?
- **RQ3**. How does the age of the software system affect the interest?
- **RQ4**. Is there a relationship between the wasted time during the software lifecycle and the frequencies of encountering compromised quality attributes?

The remainder of this paper is structured in seven sections, where the next section introduces related work. In the third section, the research method is described. The fourth section presents the results that are discussed in section five. Finally, in section six, threats to validity are presented, and Section seven concludes the paper.

10.2. Related work

The research addressing the negative effects and impacts of TD is relatively well represented within academia [101], but there are no academic research surveying practitioners on the software quality issues in relation to different TD types, and the amount and type of paying interest in relation to the age of the software.

10.2.1. Software Quality Attributes

It is evident that software having TD, can negatively affect several different quality attributes and that the attributes can affect the software in different ways, and the level of

المنارة للاستشارات

www.r

impact can also change during the software lifecycle. Li et al.'s [15] systematic mapping study shows that most examined studies argue that TD negatively affect the maintainability and that other quality attributes are only mentioned in a handful of studies each.

According to the result presented in section IV.A, this study will mainly focus on the five quality attributes; *performance efficiency, reliability, maintainability, reusability,* and *the ability to add new features*.

Performance efficiency refers to the responsiveness of an application [33] relative to the amount of resources used under stated conditions. *Reliability* focuses on the degree to which a system performs specified functions under specified conditions for a specified period of time[140]. *Maintainability* states the degree of effectiveness and efficiency with which a system can be modified by the intended maintainers. According to ISO/IEC's classification [140], *Reusability* is a sub-category of maintenance and refers to the degree to which an asset can be used in more than one system, or in building other assets[140]. The ability to add new features to a system is somewhat expressed in ISO/IEC 25010 standard [141] as a part of maintainability stated as "*modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications",* however, in this study, we have deliberately separated it from maintainability in order to get a more in-depth understand and will further use the classification *ability to add new features*.

10.2.2. Prior research on TD related interest

Falessi et al. [104] argue that the interest is non-linear and has a probability of growing exponentially rather than linearly. By analyzing source code, Nugroho et al. [142] describe that the interest grows differently on a 10-year horizon, depending on a star rating system of the software. Kazman et al. [116] have quantified the architectural sources of TD within a study and states that this TD type incurs high maintenance penalties. However, this study does not focus on the other type of negative effects in terms of other quality issues. Although some research has been done on addressing the growth of the interest, there is a lack of empirical investigations quantifying TD from a holistic perspective (including all types of TD) and describing the interest, in terms of compromised quality attributes, in more detail.

10.3. Methodology

This study is conducted using a combination of quantitative and qualitative research methodologies and by using methodological, source and observer triangulation. Methodological triangulation refers to the utilization of both qualitative and quantitative methods for gathering data [89], source triangulation refers to using different sources of data (both interviews and survey). Observer triangulation refers to using more than one observer in the study and was performed during the analysis parts where all three researchers discussed until an agreement was reached. Triangulation is important to increase the precision of empirical research and thus to provide a broader picture.

المنارة للاستشارات

www.m

The research was conducted in three separate phases. The first phase consisted of start-up group interviews with practitioners from companies with an extensive software development. In the second phase of the study, a web-survey was constructed, distributed and data was collected with the aim of assessing and quantifying how TD has a negative effect on the quality issues identified in the first phase. In the third phase, follow-up group interviews were conducted to evaluate the awareness of the relationship between TD and compromised quality attributes and to assess the presence or absence of addressing these quality issues within the companies' software development strategy.

10.3.1. Survey

The web-survey was hosted by an online survey service called SurveyMonkey. The survey was using a mix of open- and closed-ended question. The questions were a combination of optional and mandatory nature. To avoid bias in the survey, the questions were developed as neutral as possible, ordered in a way that one question did not influence the response to the next question, and a clear, unbiased instruction was provided when needed. The first draft of the survey was tested by four industrial practitioners and by two Ph.D. candidates in order to evaluate the understanding of the questions and the usage of common terms and expressions. The survey was made accessible between February and March 2016, and a reminder was sent out after two weeks to those who had been specifically invited. The survey was anonymous, and participation in the survey was voluntary.

10.3.1.1. Data Collection

The survey invitation was mailed to seven companies/partners within our networks (all located in Scandinavia, with an extensive range of software development), and invitations were also published at software engineering related networks on LinkedIn. Across all these collaborators, 312 respondents began the survey, and 258 respondents answered all questions (a completion rate of 83%). This paper's response rate will refer to the participants who started and completed the survey.

The first part of the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their companies. This data is compiled and presented in Table 1. The level of education of the respondents was quite high, with 58% having a Master degree and 25% having a Bachelor degree. The most common size of the software development team included 6-10 members (36 %), and most (32%) systems were on average 5-10 years old from their initial design. Nevertheless, a significant number (35%) of the respondents' systems were more than ten years old.

The second part of the survey included questions based on the research questions presented in Section 1. In this part of the survey, following survey questions were asked (SQ):

- SQ1: Which of the following challenges generates the most negative impact on your daily software development work? Please rank them from 1 to 11.
- SQ2. How much of the overall development time is wasted because of these issues?

المنارة للاستشارات
www.r

- SQ3. During the software life cycle, you might encounter some of the following issues. How often do you experience them?

For the question SQ1, different categories and sub-categories of TD (Complex Architectural Design, Requirement TD, Testing TD, Source Code TD, Documentation TD, Too many different patterns and policies, Dependency violations, Infrastructure TD, Lack of reusability in design, Dependencies to external resources/software, Uneasy/Tensed social interactions between different stakeholders) provided by [15],[143] were used as answering alternatives. To make sure that the respondents were considering the correct type of TD issues, a short description of the types were used in accordance with the related work i.e. Social Debt was described as "Uneasy/Tensed social interactions between different stakeholders".

In survey questions, SQ2, these same types of TD were referred to in order to ensure a better construct validity of the survey. In SQ2, the estimation of the interest as wasted time was indicated in predefined percentage intervals of <10%, 10-20%....80-90% and I don't know.

In SQ3, the respondents could indicate their frequency of experiencing the quality issues on a 5-point Likert Scale (Very Frequently, Frequently, Occasionally, Rarely, and Never).

10.3.1.2. Data analysis

The data from the survey was analyzed in a quantitative fashion, by interpreting the numbers obtained from the answers. To allow for ordinal comparison, a transforming from Likert scale to numeric value have been used. The assigned ordinal values for the Likert scale categories are: Never=1, Rarely=2, Occasionally=3, Frequently=4 and Very Frequently=5. The data was analyzed by examining the median, mean and standard deviation and also by using the statistical methods Pearson's R, the Pearson chi-square tests, and ANOVA.

10.3.2. Interviews

10.3.2.1. Data collection

As suggested by Runeson and Höst [73], this study employed the technique of semi-structured interviews where the questions were planned but not necessarily asked in the same order as they were listed.

10.3.2.2. Start-up interviews.

Six companies were group interviewed within total 43 practitioners (developers, product owners, architects) with the aim of understanding which of the quality attributes listed in the ISO/IEC's classification [140] were the most negatively affected by having TD. These companies also participated in the survey and in the follow-up interviews. The assessment of the compromised quality attributes was done by an in-depth analysis by examining

المنارة للاستشارات

www.r

nine different TD issues and evaluating the impact each of these had on the listed ISO/IEC quality attributes.

TABLE 1. CHARACTERISTICS OF THE SURVEY PRACTITIONERS

Experience		**Education**		**Roles**	
< 2 years	3.90%	Master's degree	57.80%	Developer	49.20%
2 - 5 year	10.50%	Bachelor's degree	25.20%	Software Architect	24.80%
5 - 10 year	17.40%	No Univ. education	6.20%	Manager	6.20%
> 10 years	68.20%	Other:	5.80%	Project Manager	6.20%
		Ph.D. degree	5.00%	Product Manager	5.0%
Gender				Expert	5.0%
Male	89.90%	**Software system type***		Tester	3.50%
Female	8.50%	Embedded system	48.84%		
Other	1.60%	Real-time system	41.86%	**Team size**	
		Data mgn system	22.09%	1-5 members	23.30%
System Age		System Integration	20.93%	6-10 members	36.00%
< 2 years	9.70%	Modeling and/or simul.	15.12%	11-20 members	15.90%
2-5 years	23.3%	Data analysis system	14.73%	21-40 members	6.60%
5-10 years	32.2%	Web 2.0 / SaaS system	8.53%	> 40 members	18.20%
10-20 years	28.3%	Other	8.53%		
>20 years	6.6%				

10.3.2.3. Follow-up interviews.

In total, we group interviewed seven companies where each interview included between 4 to 7 participants. Altogether, we interviewed 32 experienced software development professionals with roles such as architects, developers, product owners and managers. Each interview lasted between 105 and 120 minutes and was digitally recorded and transcribed verbatim. During the interviews, compiled results from the previous survey were presented to the respondents, where some of the most interesting findings were highlighted together with questions related to the specific area of research.

10.3.2.4. Data analysis

The transcriptions from the recorded interviews were manually coded. To keep track of the links between the codes and the quotations, the coding was supported by using a Qualitative Data Analysis (QDA) tool. Based on the research taxonomy, a coding scheme containing corresponding codes and sub-codes were implemented. To ensure that the coding was done as consistently and reliably as possible, two authors individually coded a set of codes and sub-codes and thereafter a synchronization of the output was performed.

10.4. Results and findings

The following subsections present results for the research question presented in Section I. The results from RQ1 are used as input in the next following research questions.

المنارة للاستشارات

www.r

10.4.1. Affected Quality Attributes

Which quality attributes are the most affected by Technical Debt and how frequent are those encountered during the software lifecycle and are those issues explicitly address within software companies? (RQ1)

The aim of the six initial startup-interviews was to obtain a more profound view of which software qualities were negatively affected and compromised due to TD. During these startup-interviews, each ISO/IEC quality attribute presented in Section 2.1, was first discussed separately. Secondly, nine different TD issues were analyzed in order to evaluate which quality attributes were most negatively affected by the TD issue. As a result from these startup-interviews, we got an aggregated view of the most compromised quality attributes, and it was evident that *maintainability, reliability, performance, reusability* and *the ability to add new features* were the most frequently compromised software qualities due to TD in the software.

These five quality attributes were then used in the survey, for examining the frequency of encountering them among all 258 survey participants. The summary statistics from the survey showed that 60 % of all the respondents frequently or very frequently encounter maintenance difficulties during the software lifecycle and 45 % of the respondents frequently or very frequently encounter restricted reusability and 39 % of the respondents frequently or very frequently encounter a limited ability to add new features. Another interesting finding was that 3.5% (7 developers, one manager, and one software architect) of the respondents indicated that they never experienced performance degradations in their software, compared to that only one respondent reported never experiencing maintenance difficulties.

The majority of the interviewees confirmed that TD, in general, has a major and direct negative effect on those quality issues. However, one individual stated that restricted reusability was not in direct focus within their strategy *"We find that it is sort of expensive to build generic functions so that they can be reused, it is too expensive in some cases. So for us, it becomes less and less important actually.....We use the experiences from different parts, but we do not reuse the code as much as we used to."*

10.4.2. TD Types and related interest

What type of TD generates the most negative impact on daily software development work and what kind of interest is related to each TD type? (RQ2)

To understand what type of TD generates the most negative effect, the respondents were asked to rank (score 1 = lowest impact, score 11 = highest impact) a randomly ordered list of 11 different TD types which they consider to have the most negative impact on their daily software development work (one rank for each type). These estimations reflect the respondents' perceptions of the negative impact of each TD type.

To make it easier to understand, analyze and compare the different TD types, the ranking scores were broken down into smaller subintervals, where the highest scores 9-11 are aggregated as High, score 4-8 are aggregated as Medium and scores below 4 are aggregated as Low.

المنارة للاستشارات

www.r

In Table 2, the frequency for each TD type is tabulated and compared by the aggregations of three different levels of negative impacts on the daily software development work. The cells in the table are colored, to make it easier when comparing the values (the grayer the cell, the higher the frequency is). As it can be seen from Table 2, Complex Architecture Design (42.2%), Requirement TD (40.3%), and Testing TD (37.6%) reported significantly increased frequencies in the highest impact interval, compared to the other listed TD Types. To examine how frequently each of the compromised software quality issues is encountered for each TD type, an assessment of the mean value of each quality attributes in the high-frequency interval is examined.

TABLE 2. FREQUENCY OF EXPERIENCE OF EACH TD TYPES

TD Type	Frequency (%)		
	High	***Medium***	***Low***
Complex Architectural Design	42,2	38,0	19,8
Requirement TD	40,3	38,4	21,3
Testing TD	37,6	34,1	28,3
Source Code TD	29,1	38,4	32,6
Infrastructure TD	25,2	29,8	45,0
Documentation TD	24,8	41,9	33,3
Dependencies to external resources/software	22,5	26,7	50,8
Too many different patterns and policies	21,3	41,1	37,6
Uneasy/Tensed social interactions between different stakeholders	19,4	24,8	55,8
Lack of reusability in design	19,0	39,1	41,9
Dependency violations	18,6	47,7	33,7

المنارة للاستشارات

www.r

TABLE 3. FREQUENCY OF COMPROMISED SOFTWARE QUALITY

TD Type	Frequency (mean)				
	Maintenance difficulties	***Poor Reliability***	***Restricted Reusability***	***Performance degradations***	***Limited ability to add new features***
Complex Architectural Design	3,77	3,19	3,51	3,19	3,65
Requirement TD	3,64	3,04	3,57	2,96	3,39
Testing TD	3,61	3,36	3,36	2,99	3,29
Source Code TD	3,85	3,49	3,62	3,13	3,6
Infrastructure TD	3,53	3,13	3,27	3,11	3,30
Documentation TD	3,70	3,25	3,51	2,92	3,48
Dependencies to external resources/software	3,51	2,90	3,24	3,01	3,26
Too many different patterns and policies	3,67	2,79	3,25	2,96	3,28
Uneasy/Tensed social interactions between different stakeholders	3,48	2,76	3,17	2,85	3,37
Lack of reusability in design	3,6	3,13	2,91	3,13	3,29
Dependency violations	2,85	3,02	3,44	2,98	3,78

Table 3 presents a cross-tabulated view of the mean value of the encountered frequency of each TD type and each quality issue. The table is quite revealing in several ways. First, this table demonstrates that maintenance difficulties are most frequently encountered by the respondents, except for the TD type where Dependency violations are the most challenging issues (cells colored in dark gray).

Secondly, the overall highest and the lowest mean values are highlighted in Table 3 (in bold). Maintenance difficulties are most frequently encountered by respondents estimating that Source Code TD has the most negative impact on the daily work (mean 3.85). The least frequently encountered quality issue is poor reliability issues by respondents having a high negative impact of social debt (mean 2.76).

Thirdly, for the first six TD types causing the highest negative impact on the respondent's daily work, performance degradations are the least frequently encountered during the software life cycle (cells colored in light gray). Pearson correlation coefficients were calculated to study if there was a linear relationship between the frequency of

المنارة للاستشارات
www.r

encountering each quality issue and the level of negative impact of each TD type. We found a significant, correlations in Complex Architectural Design and limited ability to add new features (r (255) =.130, p =.037). This result implies that there is a linear relationship between the level of the negative effect of the Complex architectural design and how often the respondents encounter a limited ability to add new features.

10.4.3. Age of the Software

How does the Age of the software system affect the interest? (RQ3)

There is a generally held view that the negative effects of TD increase with the age of the software. In order to evaluate this view, a statistical cross-tabulated analysis showing the interest in relation to the software age interval was used. Fig. 1 captures the difference between the frequencies of encountering each software quality issue in relation to the age of the software. To evaluate if there is a significant relationship between the age of the system and the encountered frequency of each quality issue, we calculated the Pearson correlation coefficient for each pairwise combination. This test did not indicate any significant linear correlations within any time interval for any quality issue. Supplementary, a Pearson chi-square test for independence of each challenge was calculated. This test showed that there are not any significant relationships between any of the quality issues and the age of the system. When tested explicitly if the quality issues were encountered significant differently in relation to the age of the software, a one-way ANOVA test showed somewhat surprising that there were only significant differences in how the respondents encountered maintenance difficulties ($F(4,252)= 2.967$, $p=0.033$). What is striking about the data in Fig 1, in this table is that *all* of the quality issues are frequently encountered for systems with age less than 2 years. This finding shows that the payment of interest is early introduced and persist during the whole software lifecycle. We did not find any statically evidence that the negative effects of TD increase with the age of the software.

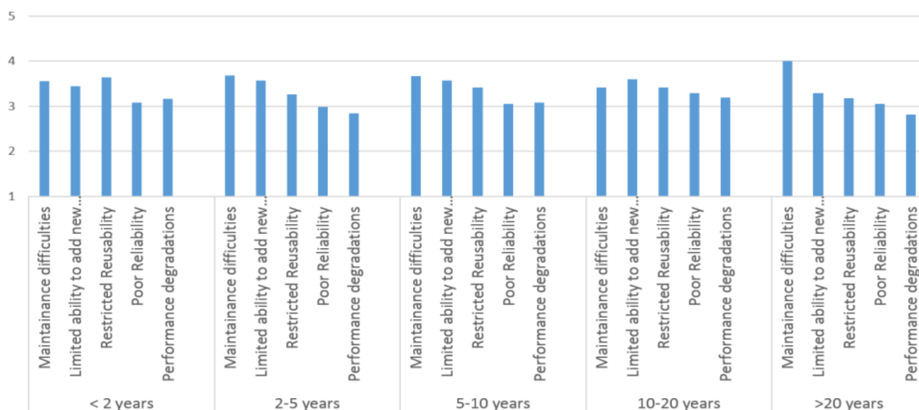


Figure 1. Frequency of each software quality issue (mean) per age interval

المنارة للاستشارات
www.r

10.4.4. Wasted time and compromised quality issues

Is there a relation between the wasted time during the software life cycle and the frequency of encountering compromised quality issues? (RQ4)

This result is based on a cross-tabulated analysis for each of the investigated quality issues for those respondents who *frequently or very frequently* encounter each of the quality issues and their estimated wasted time.

As illustrated in Fig. 4 there is a clear trend of increasing wasted time in relation to how frequently each quality issue is encountered by the respondents. What stands out in this figure are that 100 % (11 respondents) who frequently or very frequently encounter a *limited ability to add new features* estimate their overall waste of time during the lifecycle to more than 80 %. Based on Pearson R correlation, table 4 shows that there is a significant positive linear correlation between all of the investigated quality issues and the amount of the estimated wasted time. The strongest relationship was found between the frequency of encountering *poor reliability* and wasted time, follow by a *limited ability to add new features* and *maintenance difficulties*.

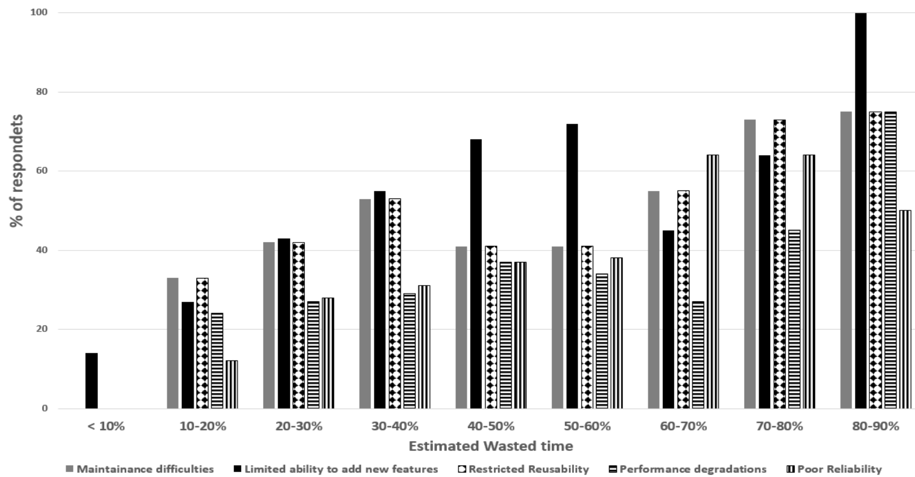


Figure 2. The frequency of encountered (frequently or very frequently) each quality issues in relation to the overall wasted time during the software lifecycle.

TABLE 4. THE RELATION BETWEEN WASTED TIME AND QUALITY ISSUE

Pearson R Correlation			
Quality issue	**ρ**	**P-value**	**Strength of correlation**
Maintenance difficulties	0,262	0,000	Medium - Low
Limited ability to add new features	0,288	0,000	Medium - Low

المنارة للاستشارات
www.r

Pearson R Correlation			
Quality issue	**ρ**	**P-value**	**Strength of correlation**
Restricted reusability	0,159	0,013	Low
Performance degradations	0,154	0,015	Low
Poor reliability	0,313	0,000	Medium

10.5. Discussions and Limitations

10.5.1. Affected Quality Attributes

Which quality attributes are the most affected by TD and how frequent are those encountered during the software lifecycle and are those issues explicitly address within software companies? (RQ1)

Except for maintainability issues, very little was found in the literature on the question of which quality attributes are most commonly affected by TD and the impact of each of them. During the startup-interviews, it was evident that the respondents considered TD is causing quality issues in general, and the interviewees recognized specifically the negative impact on *maintainability, reliability, performance, reusability* and *the ability to add new features*. This analysis was also confirmed during the follow-up interview where the majority of the interviewees' company had identified these issues and addressed them in their software strategy.

When examining the frequency of encountering these quality issues within the survey, the most obvious finding to emerge from the analysis was that *all* of these quality issues were on average more often than occasionally encountered whereas, maintenance difficulties were the most frequently encountered quality issue. This finding supports the importance of that software companies need to manage TD and reverse TD proactively or potentially face compromised quality issues in the future.

10.5.2. TD Types and related interest

What type of TD generates the most negative impact on daily software development work and what kind of interest is related to each TD type? (RQ2)

The second question in this study sought to determine the level of negative impact different TD types have in daily development work and to relate each of these TD types to the frequency of encountering the identified quality issues. The overall survey results reveal that Complex Architectural Design, closely followed by Requirement TD, generates the most negative impact on daily software development work. When conducting a more detailed analysis of all the different TD types concerning the quality issues, *maintenance difficulty* was found as most frequently encountered within all

المنارة للاستشارات

www.r

different TD types, except for the TD type where Dependency violations are the most challenging issues. Perhaps one of the most important findings in this research question was the linear correlation between the level of the negative effect of the Complex architectural design and how often the respondents encounter a limited ability to add new features. However, it should be noted that a significant correlation does not necessarily imply causation. The research presented here confirms that it is important to pay extra attention to the architectural design of the software, in order for the software to be able to grow in the future.

10.5.3. Age of the Software

In what way does the Age of the software system affect the interest? (RQ3)

There is a general view that the quality issues such as maintenance complications, restricted usability, limited ability to add new features, performance degradations and poor reusability increase over time within the software lifecycle. However, the findings of this study do not support such views.

The results show no linear or any other significant relationships between the age of the system and the frequency of encountering each of the quality issues. The frequency of encountering the quality issues within the different time interval was only significantly different for maintenance difficulties and for the other quality issues, no significant differences were found.

10.5.4. Wasted time and compromised quality issues

Is there a relation between the wasted time during the software lifecycle and the frequency of encountering compromised quality issues? (RQ4)

These research findings confirm that there are significant positive linear correlations between *all* of the investigated quality issues and the amount of the estimated wasted time. This indicates that the more often the practitioners encounter each quality issue, the more software development time is wasted.

This indicates that the more software development time is wasted, the more often the practitioners encounter each quality issue. Taken together, these results suggest that software productivity in terms of less wasted time could be increased by remediating TD and thereby less frequently encountering the quality issues and in that way, waste less software development time.

Taken together, these findings raise novel insights about how the consequences due to having TD, negatively affects *several* different software quality attributes. Based on collected data from 258 software practitioners, the generally held view of that the frequency of encountering software quality issues increases in relation to the age of the software, was not possible to confirm. Contrary, our result shows that the frequency of encountering the compromised quality attributes starts early and continuous during the whole lifecycle. The results also imply that by remediate TD will gain less software quality issues.

المنارة للاستشارات

www.r

10.6. Threats to validity

For verifiability reasons, we have made information available online. All survey questions, used in this paper, are available at the link

https://zenodo.org/record/163116#.WA9A3_5PpD8.

The result of this research may be affected by some threats to validity. *Construct validity* reflects what extent the operational measures that are studied represent, what the researchers have in mind and what is investigated according to the research questions [73]. To mitigate this risk and to make sure that the respondents were considering the correct type of TD issues, a short description of each type of TD was used in the survey. An additional threat to this validity can stem from the fact that the qualitative data derived from the survey are based on perceptions and estimations (not on measured or observed data) made by the respondents. This study could suffer from *internal validity* when the causal relationships were examined as it affects our ability to explain the phenomena that we observed accurately. Thus, our findings are correlational, and we cannot infer causality, an example of this could be where we use the estimated wasted time correlated with the frequency of how often the respondent encountered each of the compromised quality attributes. By doing this correlation, this validity could have been violated by either finding relationships that are nonexistent or missing real relationships that are wrongly deemed non-significant. However, taken together, the findings indicate a causal relationship but to needs to be better verified with further studies. *External validity* focuses on to what extent it is possible to generalize the findings. There is always a risk in surveys that the sample is biased and for this topic, a potential threat refers to the demographic distribution of response samples. Most of the data from the invited companies for the survey and the interviewed companies was gathered in Scandinavia. Thus, the results might be different in other geographical and cultural areas. *Reliability* addresses whether the study would yield the same results if other researchers replicated it. To mitigate this threat, we have employed source triangulation, methodological triangulation, and finally observer triangulation.

10.7. Conclusion

Technical Debt (TD) is evidently detrimental to software companies, and it is important to assess and estimate the consequences of TD in terms of its negative impact on system quality attributes. This study is based on initial startup interviews with 43 participants, a survey with 258 respondents and follow-up interviews with 32 practitioners. This study has shown that TD most negatively affects the quality attributes maintainability, reliability, performance, reusability and the ability to add new features, where maintenance difficulties, restricted reusability and limited ability to add new features are most frequently encountered quality issues during the software lifecycle.

This study also reveals a linear relationship between the level of the negative effect of complex architectural design and the frequency of how often the respondents encounter a limited ability to add new features. This study could not confirm the generally held view that the amount of compromised quality attributes increases with system age, our results

المنارة للاستشارات

www.r

imply that it is important to very *early* in the lifecycle remediate TD in order to the keep the frequency of compromised quality attributes down.

This study has also shown that there is a correlation between how software practitioners estimated their wasted software development time and the frequency of how often they experience each quality issue.

All these findings emphasize the importance of understanding how TD negatively affects the system quality, in order to proactively manage it in terms of allocating time, resources and additional effort. These findings provide strong empirical confirmation that both practitioners and researchers need to focus more attention and effort on deliberately remediating TD, in order to reduce costly interest payments.

المنارة للاستشارات

www.r

المنارة للاستشارات
www.m

11. The Pricey Bill of Technical Debt

The aim of this chapter is to estimate waste of working time, caused by the TD during the software lifecycle, and also to investigate how practitioners perceive and estimate the impact of the negative consequences due to TD during the software development process.

Software companies need to support continuous and fast delivery of customer value both in short and a long-term perspective. However, this can be hindered by evolution limitations and high maintenance efforts due to internal software quality issues by what is described as Technical Debt. Although significant theoretical work has been undertaken to describe the negative effects of Technical Debt, these studies tend to have a weak empirical basis and often lack quantitative data. This paper reports the results of both an online web-survey provided quantitative data from 258 participants and follow-up interviews with 32 industrial software practitioners. The importance and originality of this study contributes and provides novel insights into the research on Technical Debt by quantifying the perceived interest and the negative effects it has on the software development lifecycle. The findings show that on average, 36 % of all development time is estimated to be wasted due to Technical Debt; Complex Architectural Design and Requirement Technical Debt generates most negative effect; and that most time is wasted on understanding and/or measuring the Technical Debt. Moreover, the analysis of the professional roles and the age of the software system in the survey revealed that different roles are affected differently and that the consequences of Technical Debt are also influenced by the age of the software system.

This chapter has been published as:
The Pricey Bill of Technical Debt – When and by whom will it be paid?
T. Besker, A. Martini, and J. Bosch
In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, pp. 13-23, 2017.

11.1. Introduction

Technical Debt (TD) is recognized as a critical issue in today's software development industry [7]. Since this phenomenon is detrimental to software companies, it is important to assess and estimate the consequences of Technical Debt in terms of the scale of wasted time and effort.

Ward Cunningham [91] introduced the financial metaphor of TD in order to describe to nontechnical product stakeholders the need to recognize the potential long-term negative effects of the immature code that is made during the software development lifecycle. Such debt has to be repaid with interest in the long term. Interest is the negative effect of the extra effort that has to be paid due to the accumulated amount of TD in the system, such as executing manual processes that could potentially be automated or spending excessive effort on modifying an unnecessarily complex code, performance problems due to lower resource usage caused by an inefficient code and similar costs. Ampatzoglou et al. [30] define interest as "*The additional effort that is needed to be spent on maintaining the software because of its decayed design-time quality.*"–Nowadays, it is well established

المنارة للاستشارات
www.m

that TD has a negative impact on software development organizations by hindering evolution and causing high maintenance costs due to internal software quality issues [18],[109],[22],[24],[103].

Left unmanaged, TD can result in unexpectedly large cost overruns, severe quality issues, inability to add new features [50] and even lead to a crisis point when a huge, costly refactoring or the replacement of the entire system needs to be undertaken [5],[95].

There are many different types of TD (such as Architectural TD, Requirement TD, Test TD, Code TD), which differ in the degree of their negative impact and consequently cause various levels of wasted time during the software development process. Several professional roles participate in the software development process, and could possibly be affected by TD in diverse ways. Furthermore, as the software ages, different types of TD could have varying negative effects and could possibly generate a dissimilar distribution of extra time spent on different activities. However, little knowledge and few supporting tools are available to measure the extent of TD within a system and, in addition, the time spent on TD related issues is not made explicitly visible and measurable. Without such knowledge, software development organizations do not know the interest that they are paying on the debt, and therefore they might not give TD management the necessary attention.

We will answer the research questions by using survey data based on software professionals' perceptions. All survey respondents were experienced in software development, and therefore, their estimates were likely to be formed by what they have heard, observed, and experienced at their workplaces.

To the best of our knowledge, there are no studies quantifying the interest in terms of how much time (observed, measured or estimated) is wasted due to TD and how this wasted time varies in relation to the system age and its impact on a range of professional software roles. We, therefore, aim to answer the following research questions (RQ):

RQ1. *How much of the overall development time is wasted because of Technical Debt?*

RQ2. *What kind of Technical Debt Challenges generates the most negative impact?*

RQ3. *What are the various activities on which extra-time is spent as a result of Technical Debt?*

RQ4. *In what way does the age of the software system affect the questions stated in RQ1-3?*

RQ5. *In what way are the different software roles affected by the questions stated in RQ1-3?*

This study offers a contribution to research, with respect to the existing body of knowledge about TD in the following important areas:

1. We provide a survey- and interview-based study on how practitioners experience TD within the software industry based on both quantitative and qualitative data.
2. This study attempted to quantify the interest in terms of wasted software development time. We present results that the average estimated wasted software development time is 36%, because of TD.

المنارة للاستشارات
www.n

3. We present results showing that Complex Architectural Design, closely followed by Requirement TD, generates the most negative impact on daily software development work.
4. This study provides new insights into TD research by showing that the most wasted time is spent on Understanding and/or Measuring the TD.
5. This research reveals that different roles within software development are affected differently by TD. We found that the amount of estimated time spent as interest of TD is supported by all roles. Furthermore, different roles waste time on different activities, hence experiencing different negative impacts of TD.
6. This study shows that the degree of wasted time varies with the age of the software. Although the amount of wasted time result does not show a linear progression, the wasted time varies in relation to the system age.
7. To both practitioners and academics, this study demonstrates the relevance of paying more attention and effort to remediate TD deliberately.

The remainder of this paper is structured in seven sections, where the second section introduces related work. In the third section, the research method is described. The fourth section presents the results that are discussed in section five. Finally, in section six, threats to validity are presented, and section seven concludes the paper.

11.2. Related work

This section presents related research, both based on empirical survey-based data and on other TD related studies.

11.2.1. Empirical survey-based studies on TD

By reviewing other survey-based studies focusing on TD, we found four survey-based studies which somewhat related to our investigation.

Ernst et al. [16] conducted a survey within three large organizations, with 536 respondents and seven follow-up interviews. The result of that study shows that "bad architecture choices" are the greatest source of TD. This study also concludes that the degree of architectural drift is related to system age. They found a weak association between system age and the perceived importance of architectural issues where 89% of the respondents with system age $\geq$ 6 years agreed or strongly agreed with the notion that architectural issues were a significant source of debt, compared to 80% of those with newer. In contrast, to that study, our study also includes how different *roles* perceive TD and also has a finer granularity of the age intervals and therefore, provides a more detailed analysis. Furthermore, this study provides a wider coverage in terms of including an investigation of how ***all*** different TD issues, not only the architectural related TD, vary in relation to the system age.

[17] conducted a survey of 54 Finnish software practitioners investigating the level of TD knowledge, how TD occurs in projects and which of the applied agile components of the development was sensitive to TD. They found that the most frequent causes of TD were

المنارة للاستشارات

www.r

inadequate architecture and inadequate documentation. In this study, we cannot find a quantification or estimation of the interest, there is no explanation about what type of activities on which extra-time is spent, and perspective of different roles or age of the software are not investigated.

This paper is somewhat also related to our previous papers [144],[95], where we specifically study the *architectural type of* TD and address how TD negatively affects different *software quality* attributes. However, even though the data were collected using the same survey, this study uses different data and focusing on several different types of TD and compare those with each other regarding the scale of wasted time and effort, rather than quality.

11.2.2. Prior research on TD related issues

Kazman et al. [116] present a case study of identifying and quantifying architectural debts in an industrial software project, using code changes as a proxy for calculating the interest. This study focuses on identifying the architectural roots of TD, meaning that the study does not address the cost of interest, but instead aims at quantifying the expected payback for refactoring. These costs are to some extent based on estimated values from the interviewed architects and based on these assumptions, the expected benefit from the refactoring is calculated. Compared to that study, this study focus on several different types of TD (not only on the architectural TD) and this study have a focus on quantifying the interest in terms of wasted time instead of a focus on locating the roots of the architectural TD and the expected benefit from the refactoring. Falessi et al. [104] argue that the interest is non-linear and has a probability to grow exponentially rather than linearly. By analyzing source code, Nugroho et al. [142] describe that the interest grows differently on a 10-years horizon, depending on a star rating system of the software. They present calculations based on assumed values and empirical data which illustrates that the interest grows linear for 3-star systems and close to linear for 4-star systems. However, such study is based on known metrics (which do not cover the full spectrum of TD types) and on code changes as an estimation of interest.

In summary, although, there have been some attempts on quantifying the interest of TD, there is a lack of empirical investigations including software practitioners. Also, TD interest has not been studied holistically (including all types of TD). Finally, the interest has not been put in relation to the system age and to how it affects different professional roles.

11.3. Methodology

In this study, a combination of quantitative and qualitative research approaches is used. We aimed at using methodological, source and observer triangulation in order to increase the validity and the reliability of the result. Methodological triangulation refers to the utilization of both qualitative and quantitative methods for gathering data [89], source triangulation refers to using different sources of data while observer triangulation refers using more than one observer in the study [73]. Triangulation is important to increase the

المنارة للاستشارات

www.m

precision of empirical research and thus to provide a broader picture [73]. The following sections describe the methods used for conducting the survey and the complementing qualitative follow-up interviews.

11.3.1. Survey

The web-survey was designed and hosted by the online survey service called *SurveyMonkey*. The survey was using a mix of open- and closed-ended questions. The questions were a combination of optional and mandatory nature. To avoid bias in the survey, the questions were developed as neutral as possible, ordered in a way that one question did not influence the response to the next question, and a description was provided when needed [78]. The first draft of the survey was tested by four industrial practitioners (developer, manager, project owner, and software architect) and by two Ph.D. candidates in order to evaluate the understanding of the questions and the usage of common terms and expressions [77]. During this evaluation, we also monitored the time needed to complete the survey. The survey was made accessible between February and March 2016, and a reminder was sent out after two weeks to those who had been specifically invited. The survey was anonymous, and participation in the survey was voluntary.

11.3.1.1. Data collection

The survey invitation was mailed directly to seven companies/partners within our networks, all located in Scandinavia, with an extensive range of software development, and invitations were also published at software engineering related networks on LinkedIn. Across all these collaborators, 312 respondents began the survey, and 258 respondents answered all questions. Due to this high completion rate (83%), we decided to reject the incomplete questionnaires, according to the guidelines by Kitchenham and Pfleeger [78].

The first part of the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their companies. This data is compiled and presented in Table I. The level of education of the respondents was quite high, with 58% having a Master degree and 25% having a Bachelor degree. The survey included respondents having different roles, where 49% were Developers/Programmers/-Software Engineers, while 25% were Software Architects. Approximately 78% of the Software architects and 59% of the Developers/- Programmers/Software Engineers had more than ten years of experience. The most common size of the software development team included 6-10 members (36 %), and most (32%) systems were on average 5-10 years old from their initial design. Nevertheless, a significant number (35%) of the respondents' systems were more than ten years old. Most of the software systems were embedded systems (49%) and Real-time system (42%). However, in this specific survey question, the respondents could select more than one option.

The second part of the survey included questions based on the research questions presented in Section I.

المنارة للاستشارات

www.r

In this part of the survey, the participants were asked to estimate their perception of the three survey questions (SQ):

SQ1. Which of the following challenges generates the most negative impact on your daily software development work? Please rank them from 1 to 11.

SQ2. How much of the overall development time is wasted because of these issues? (Do not consider fixing and managing the issues, just if they hinder you when you add/change/understand the system).

SQ3. If you take into consideration your most recent projects and the issues listed in SQ1, which of the following activities did you spend most of your extra time on? Please rank them from 1 to 6.

TABLE I - CHARACTERISTICS OF THE SAMPLE SURVEY

Individual		Company	
Experience		**Team size**	
< 2 years	3,90%	1-5 members	23,30%
2 - 5 year	10,50%	6-10 members	36,00%
5 - 10 year	17,40%	11-20 members	15,90%
> 10 years	68,20%	21-40 members	6,60%
Education		> 40 members	18,20%
Master's degree	57,80%		
Bachelor's degree	25,20%	**Software system type***	
No Univ. education	6,20%	Embedded system	48.84%
Other:	5,80%	Real-time system	41.86%
Ph.D. degree	5,00%	Data management system	22.09%
Roles		System Integration	20.93%
Developer/Program/ -Software Engineer **	49,20%	Modeling and/or simul.	15.12%
Software Architect	24,80%	Data analysis system	14.73%
Manager	6,20%	Web 2.0 / SaaS system	8.53%
Project Manager	6,20%	Other	8.53%
Product Manager	5,0%	**System Age**	
Expert	5,0%	< 2 years	9,70%
Tester	3,50%	2-5 years	23,3%
Gender		5-10 years	32,2%
Male	89,90%	10-20 years	28,3%
Female	8,50%	>20 years	6,6%
Other/no share	1,60%		

* More than one option was selectable, ** Abbreviated as ***Developer*** in this paper

For the question SQ1, the different categories and sub-categories provided by [15] were used to distinguish the different TD types (Complex Architectural Design, Requirement TD, Testing TD, Source Code TD, Documentation TD, Too many different patterns and policies, Dependency violations, Infrastructure TD, Lack of reusability in design, Dependencies to external resources/software, and Uneasy/Tensed social interactions between different stakeholders).

In survey questions, SQ2 and SQ3, these same types of TD were referred to in order to ensure a better construct validity of the survey. In SQ2, the estimation of the interest as wasted time was indicated in predefined percentage intervals of <10%, 10-20%....80-90%

المنارة للاستشارات

www.r

and I don't know. In order to separate the time spent on management, the respondents were asked not to include the time spent on fixing and managing the issues. In SQ3, the respondents could indicate their level of agreement on a 6-point Likert Scale (Strongly agree…Strongly disagree).

11.3.1.2. Data analysis

The data from the survey was analyzed in a quantitative fashion, i.e. by interpreting the numbers obtained from the answers. All analyses were carried out using the software SPSS (version 22). Descriptive statistics were employed to organize and analyze the qualitative data by using tables and charts. To allow for ordinal comparison, a transforming from Likert scale to numeric value has been used. The assigned ordinal values for the Likert scale categories were: Never = 1, Rarely = 2, Occasionally = 3, Frequently = 4 and Very Frequently = 5.

The data was analyzed by studying the median, mean and standard deviation and also by using the statistical methods Pearson's R, the Pearson chi-square tests, and ANOVA. The Pearson's R method computes the pairwise determining the strength and direction of the association between two metrics and can be used to measure a linear association between two metrics. The Pearson chi-squared tests are used for evaluating how likely it is that any observed difference between the sets arose by chance, and the test of independence assesses whether unpaired observations on two variables are independent of each other. The one-way ANOVA was used to identify significant differences in mean values. Mainly three different types of quantitative analysis were conducted, in order to synthesize the result for each research question:

- **General results**: all the answers are considered, composed and used as a result to draw conclusions common to all roles and all system ages.
- **System Age:** all answers are grouped into system age intervals of <2 years, 2-5 years, 5-10 years, 10-20 years, and >20 years. The time refers to the age from initial design. The age intervals are treated as categorical data.
- **Role**: all the answers are grouped into roles of Developer, Expert, Manager, Product Manager, Project Manager, Software Architect, and Tester.

11.3.2. Interviews

11.3.2.1. Data collection

Interviews can be divided into unstructured, semi-structured and fully structured interviews. As suggested in [145], this study employed the technique of semi-structured interviews where the questions were planned but not necessarily asked in the same order as they were listed. This semi-structured interview technique allowed flexibility and exploration of the studied objects by asking follow-up questions based on the respondents' answers. All interviews were group interviews and were conducted using the guidelines suggested by Krueger and Casey [146].

المنارة للاستشارات

www.r

In total, we interviewed seven companies, where each interview included between 4 to 7 participants. These companies did all participate in the survey, and all the interviewees had answered the survey before the interview. Altogether, we interviewed 32 experienced software development professionals with roles as architects, developers, product owners and managers.

Each interview lasted between 105 and 120 minutes and was digitally recorded and transcribed verbatim. During the interviews, compiled results from the previous survey were presented to the respondents, where some of the most interesting findings were highlighted together with questions related to the specific area of research. This presentation allowed the respondents to more easily relate the interview questions to the result of the survey.

The interview questions were designed to a) increase the understanding of the survey results, b) ensure that the questions in the survey were understood and interpreted as intended and in a uniform way, c) confirm the result from the survey, and d) understand the implications of the survey results. The questions were developed to cover the same taxonomies as in the survey. The following are examples of the questions asked during the interview:

- What do you consider to be a Complex Architectural Design?
- If the wasted time could be reduced, how would that affect your software development process?

11.3.2.2. Data analysis

The transcriptions from the recorded interviews were manually coded using established guidelines in the literature [147]. To assess and rate the level of confirmation by the interviewees of the survey result, a three-level classification of the confirmation was used; *Strongly Confirming* (SC), *SomeWhat Confirming* (SWC) and *DisConfirming* (DC). The result from the survey, which was expressly and unambiguously confirmed by interviewees, was classified as *SC*. The results which were not confirmed or those that were confirmed but with deviations were classified as *DC*.

11.4. Results and Findings

The following subsections present results for the research questions presented in Section I, and the results are grouped according to each research question.

11.4.1. How much of the overall development time is wasted because of Technical Debt? (RQ1).

This research question focuses on the TD interest in terms of how much of the overall development time is wasted. The estimated wasted time does not include fixing and managing the issues. Fig. 1 provides an overview of the distribution of the estimated wasted time which is represented by different percentage intervals of the overall development time.

المنارة للاستشارات

www.r

The most striking result emerging from the data is that, on average, the respondents estimated that **36% of all software development time is wasted because of paying the interest of TD** (excluding the Don't know option -3,5% of the respondents), with the standard deviation of 17,8%. It can be seen from the data in Table II that the result from the survey was almost consistently *Strongly Confirmed* (SC) during the group interviews. Only during one interview (Intv 6), one interviewee expressed "*It is probably more than 36%* ".

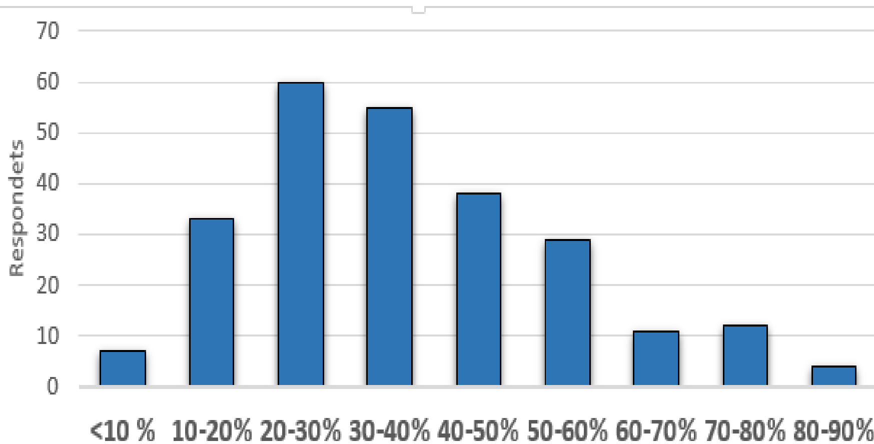


Figure 1. Estimated wasted software development time

Furthermore, the interviews also coherently revealed that if this detrimental waste of time could be reduced, the company would spend more time on adding new features and reducing the time to market for their software products. The interviewees also unanimously described that this huge waste of time is detrimental to their work, and its implications must be clearly recognized and understood.

TABLE II - INTERVIEW CONFIRMATION – WASTED TIME

Intv 1	*Intv 2*	*Intv 3*	*Intv 4*	*Intv 5*	*Intv 6*	*Intv 7*
SC	SC	SC	SC	SC	SWC	SC

11.4.2. System Age effect on wasted time (RQ4)

In order to assess whether the interest changed or not during the software life cycle, a statistical cross-tabulated analysis showing the interest in terms of the wasted time in relation to the system age interval was used. Fig. 2 captures the difference between the estimated time wasted in relation to the age of the software. For a system with age less than two years, on average, 33.8% of the software time is estimated to be wasted but for a system older than 20 years, the estimated wasted time average is 40.5%. The degree of the wasted time thus varies with the age of the software and to evaluate if there is a linear correlation between the system age and the wasted time, we used the Pearson correlation method.

المنارة للاستشارات

www.r

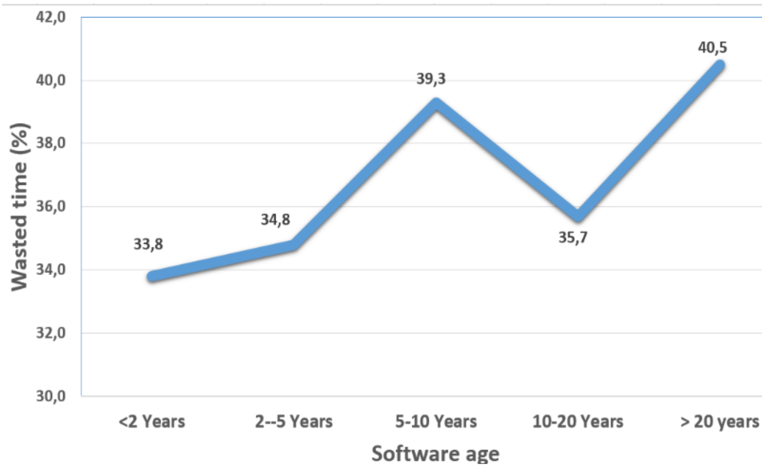


Figure 2. Wasted time in relation to System Age

These two data series (we used the average age within each year interval), returned a p-value of 0.059, which showed no support for a linear correlation of the wasted time and the age of the system. We concluded that no evidence was found for linear associations between the amount of wasted time and the age of the system.

Furthermore, the Pearson Chi-square test of dependence ($\chi 2$= 58.64, df=36, p=0.010) showed that the two factors (wasted time and system age) are not independent, which means that we could not reject the hypothesis that the system age influences the wasted time.

11.4.3. Different Roles' estimation of the wasted time (RQ5)

In this section, we explore whether different professional roles estimate the wasted time differently. When tested explicitly if the roles estimate the wasted time differently, a one-way ANOVA test showed that there were no significant differences (F(6.242)=0.926,p=0.477).

To graphically visualize the differences of distributions of the estimated wasted time, Fig. 3 represents a boxplot for each of the roles. The box-plot represents the minimum and maximum range values, the upper and lower quartiles, and the median. From the boxplot, we can see that the estimated wasted time is relatively consistent with the different roles (median range of 30-45 % and standard deviation range of 14.4–20.6 %). In this figure, we can also see that Software Architects and Experts estimate the highest average percent of the wasted time (41%), whereas Developers estimate the lowest value of wasted time (35%). One unanticipated result of the analysis showed that five of the Software Architects and seven of the Developers were found to estimate their wasted time to more than 70%.

المنارة للاستشارات

www.r

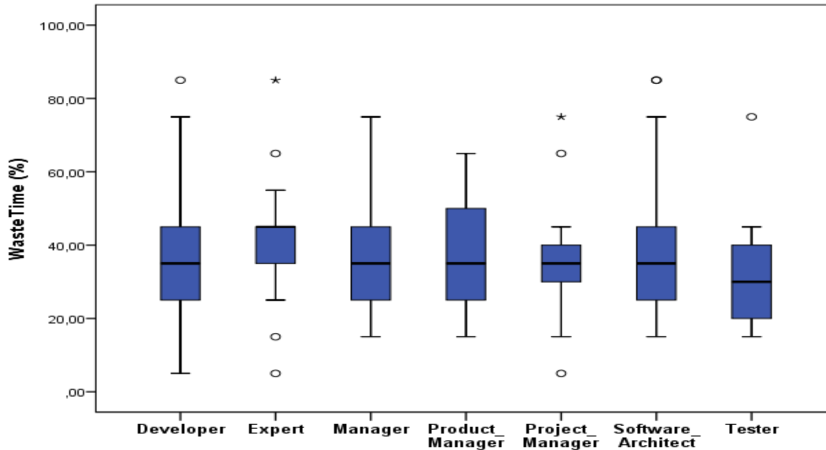


Figure 3. Distribution of the estimated Wasted Time by different Roles

11.4.4. What Challenges generate the most negative impact on the daily software development work? (RQ2)

To understand what type of TD generates the most negative effect, the respondents were asked to rank (score 1 = lowest impact, score 11 = highest impact) a randomly ordered list of 11 different *challenges* which they consider to have the most negative impact on their daily software development work. The different listed *challenges* reflected the different types of TD that emerged from [15].

There are several possible statistic values to study when evaluating a ranking result. This question involves an examination of the mean, the median and a grouping of different sets of scores.

In Fig. 4, the summary statistics for each Challenge reveal that a *Complex Architectural* Design (mean rank 7.27) and *Requirement TD* (mean rank 7.23) generates the most negative impact on daily software development work, both when studying the mean and the median values. To make it easier to understand, measure, analyze and compare the different *challenges*, the ranking scores were broken down into smaller subintervals (called class intervals), where the highest scores 9-11 are aggregated as *High*, score 4-8 are aggregated as *Medium* and scores below 4 are aggregated as *Low*.

المنارة للاستشارات

www.n

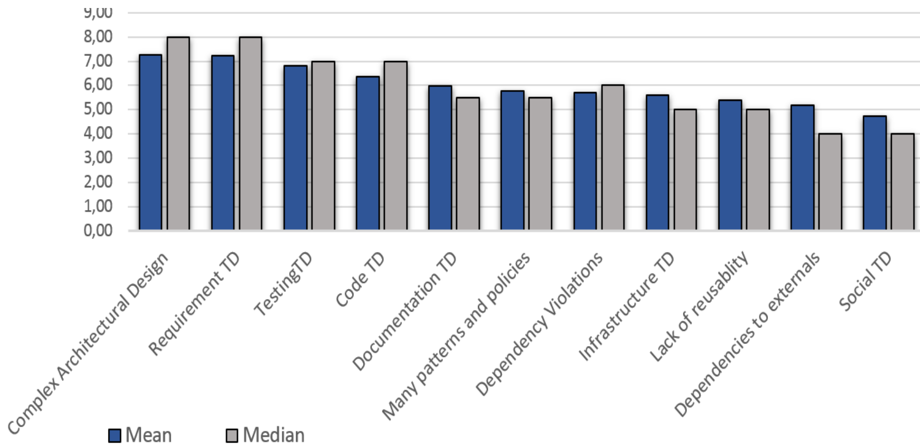


Figure 4. Mean and median of the rankings for each Challenge

For each class interval, the amount of data items falling within each impact interval is counted.

In Table III, the frequency for each *challenge* is tabulated and compared by the aggregations of three different levels of negative impacts on the daily software development work.

To make it easier when comparing the values, the cells in the table are colored (the grayer the cell, the higher the frequency is). As it can be seen from this table *Complex Architecture Design* (42.2%), *Requirement TD* (40.3%), and *Testing TD* (37.6%) reported significantly increased frequencies in the highest impact interval, compared to the other listed *challenges*.

As illustrated in Table IV, during the group interviews where this result was presented, the interviewees *Strongly Confirmed* (SC) this result as reflecting their interpretation of the *challenges* within their organizations consistently.

TABLE III - FREQUENCY OF CHALLENGES

Challenge	Frequency (%)		
	High	***Medium***	***Low***
Complex Architectural Design	42,2	38,0	19,8
Requirement TD	40,3	38,4	21,3
Testing TD	37,6	34,1	28,3
Source Code TD	29,1	38,4	32,6
Infrastructure TD	25,2	29,8	45,0

المنارة للاستشارات
www.r

Challenge	Frequency (%)		
	High	*Medium*	*Low*
Documentation TD	24,8	41,9	33,3
Dependencies to external resources/software	22,5	26,7	50,8
Too many different patterns and policies	21,3	41,1	37,6
Uneasy/Tensed social interactions between different stakeholders	19,4	24,8	55,8
Lack of reusability in design	19,0	39,1	41,9
Dependency violations	18,6	47,7	33,7

TABLE IV - INTERVIEW CONFIRMATION – CHALLENGES

Intv 1	*Intv 2*	*Intv 3*	*Intv 4*	*Intv 5*	*Intv 6*	*Intv 7*
SC	SC	SC	SC	SC	SC	SC

11.4.5. System Age effect different negative effects. (RQ4)

To assess whether the age of the system has an influence on the *challenges* during the software lifecycle, we measured the pairwise combinations for the different system age intervals with the means of the *challenges*.

Fig. 6 shows each *challenge's* impact in relation to the system age interval, that respondents of different software system ages interpret the *challenges* differently and that the impact of the *challenges* varies over time.

Determined by one-way ANOVA, there is a statistically significant difference between the mean value in the system age intervals for *Requirement TD* ($F(10)= 1.917$, $p= .043$), *Complex Architectural Design* ($F(10)= 1.928$, $p= .042$) and *Uneasy/-Tensed social interactions* ($F(10)= 1.911$, $p= .044$).

Furthermore, the data in Fig. 6 reveals that Requirement has the highest mean value for systems that are less than ten years old, whereas, for systems older than 10 years, a Complex Architectural Design has the highest mean value.

To evaluate if there are any correlations between the time intervals (where the average age within each interval is used) and the median of the scores, we calculated the Pearson correlation coefficient by measuring the strength of a linear relationship between the pair data. The Pearson correlation coefficient for all the data series varied between 0.013 and

المنارة للاستشارات
www.r

0,179 (independent of a positive or negative correlation). The test is not indicating any significant linear correlations in any time interval. Supplementary, a Pearson chi-square test for independence of each challenge was calculated to evaluate if any factors gave a significant response (p-value < 0.1). This test showed that the factors for system age and Requirement ($\chi2$= 59.33, df= 40, p= 0.025), Patterns and policies ($\chi2$= 58.8, df = 40, p = 0.028), and Social interactions ($\chi2$= 77.5, df = 40, p = 0.000) are not independent. This indicates that there is a significant relationship between these *challenges* and the system age.

The previous result showed that *Complex Architectural Design* and *Requirement TD* generate the most negative impact on daily software development work, and the value of Requirement *TD* was observed having the absolute highest negative impact within the age interval of 2-5 years (mean value 7.87). The value of *Complex Architecture Design* was observed having the highest negative impact on systems with an age interval of 10-20 years (mean value 8.08) and the lowest value for the age interval of <2 years (mean value 6.56).

11.4.6. How different Roles interpret the Challenge (RQ6)

In this section, we explore how different roles interpret which Challenge has the *most*, and *second most* negative impact on their daily software work, by a cross-tabulation of the two datasets. Table V provides the mean value and the highest and second highest rates (aggregation of score 9-11) of the *challenges* for each role, using the same aggregation as described in Section IV.B. Table V is quite revealing in several ways; it is apparent that for Developers, Product Managers, Project Managers, and Testers, the *Complex Architecture Design* has the most negative impact. Meanwhile, Software Architects and Managers interpret the *Requirement TD* to generate the most negative impact.

A striking result emerging from Table V is also that *Tester* is the role that estimates the greatest negative impact of *Complex Architectural Design*, were 56% or these professionals estimates this *challenge* is having the highest rankings (scores >8) with a mean value of 8.1. The overall result shows that *Complex Architecture Design* is the most commonly estimated challenge by having the greatest negative impact on the daily software development work among all the different roles.

However, it is also worth noting that both *Requirement TD* and *Testing TD* are frequently estimated as having the second most negative impact on respondents' daily software work.

TABLE V - ROLES INTERPRET NEGATIVE EFFECT OF CHALLENGES

Role	**Negative impacts**	
	Challenge (first and second rate)	***Mean Value***
Developer	1.Complex Architectural Design 2.Testing and Requirement	7,4 6,9/7,2

المنارة للاستشارات

www.r

Role	Negative impacts	
	Challenge (first and second rate)	***Mean Value***
Experts	1.Documentation 2.Requirement	7,1 7,2
Managers	1.Requirement and Testing 2. Complex Architectural Design and Lack of reusability	8,6/7,2 7,3/6,9
Product Managers	1.Complex Architectural Design 2. Environment and infrastructure and Lack of reusability	7,6 6,5/6,5
Project Managers	1.Complex Architectural Design 2. Testing and Requirement	8,3 7,0/7,9
Software Architects	1.Requirement TD 2. Low Code quality	7,3 6,9
Testers	1.Complex Architectural Design / to many Patterns and policies 2. Environment and infrastructure	8,1/7,2 6,9

11.4.7. What are the various activities on which extra-time is spent as a result of Technical Debt? (RQ3)

Due to TD in software systems, a lot of extra time is spent on different activities which do not deliver any direct value to the customer. In an attempt to understand how this extra time is spent, we asked the respondents to the survey to refer to their most recent project and rank different listed activities on which they spent their most extra time.

The ranking scale was set from 1 to 6, where score 1 refers to that activity of which the *least* extra time is spent while score 6 represents the activity of which *most* extra time is spent. The specified activities were recognized during previous research by interviews with practitioners as common activities within the TD research field [23]. It can be seen from the data in Table VII, that the survey results showed how 20.63% of the respondents estimated that most extra time (score 6) is spent on *Understanding and/or measuring the issues* (mean value 4.38), while 22.98% of the respondents estimate that most extra time (score 6) is spent on *Management processes* such as tracking, monitoring, and communication of the issues (mean value 3.89). Only 2.05% of the respondents pointed out that they spend most extra time on the activity of *Deciding, which issue to refactor first.*

As illustrated in Table VI, all the interviewees *Strongly Confirmed* (SC) the results from the survey on the spending of the extra-time.

المنارة للاستشارات

www.r

TABLE VI - INTERVIEW CONFIRMATION – ACTIVITIES

Intv 1	*Intv 2*	*Intv 3*	*Intv 4*	*Intv 5*	*Intv 6*	*Intv 7*
SC	SC	SC	SC	SC	SC	SC

11.4.8. System Age effect how the extra time is spent (RQ4)

To investigate if and in what way the system age of the software influence how the extra time is spent, a cross-tabulated table was created with the mean value of each activity and the system age intervals. In addition, we used Pearson correlation analysis to test the null hypothesis that the interest for each activity grows linearly in relation to the age of the system. Even if the distribution is not normal distribution (due to Shapiro-Wilk test, p>.05), we have applied the statistical test Pearson R since this method is not very sensitive to moderate deviations from normality [148].

Results obtained from the survey are visualized in Fig. 5 and in Table VII. The examination of the highest mean value for software with a system age less than two years reveals that most extra time is spent on the activity of *Understanding and/or measuring* the issues (mean value 4.2). This activity slightly increases for the system age intervals of 2-5 years (mean value 4.3), 5-10 years (mean value 4.4) and 10-20 years (mean value 4.5) and finally, for software with a system age more than 20 years the time spent on this activity decreases back to the initial mean value 4.2.

From the boxplot in Fig. 5, we can see the distribution of the rating of the time spent on Understanding *and/or measuring the issues* in relation to the system age. It shows that the median of the rating changes between 4 and 5 and that the answers are unevenly distributed among the different system age intervals. A striking observation emerging from the data comparison was that the extra time spent on the *Deciding which issues to refactor first* activity is the lowermost in all system age intervals, with a mean value varying between 2.6 and 2.9.

In order to test the hypothesis that the time spent on these activities increase in relation to the system age, we studied if and how the time spent on the different activities changed in relation to the system age.

First, the Pearson correlation coefficient for the system age intervals where the average year within each interval is used. The variables for each activity varies between 0.012 and 0.041 (independent of positive or negative correlation), thus not indicating any significant linear correlation between the time spent on each activity in relation to the system age.

Secondly, results from a one-way ANOVA revealed no significant differences among the time spent on the activity for each system age interval. We, therefore, reject the hypothesis that the time spent on these activities significantly differs or linearly increases in relation to system age.

المنارة للاستشارات

www.r

TABLE VII - EXTRA TIME SPENT ON ACTIVITIES

Activity	Mean	Score 6 (highest)
Understanding and/or measuring the issues	4.38	20.63 %
Management process of the issues (track, monitor, communicate)	3.89	22.98 %
Finding the issues	3.69	19.68 %
Refactoring the issues	3.59	15.35 %
Deciding which issue to refactor first	2.89	2.05 %
Other	2.50	15.15%

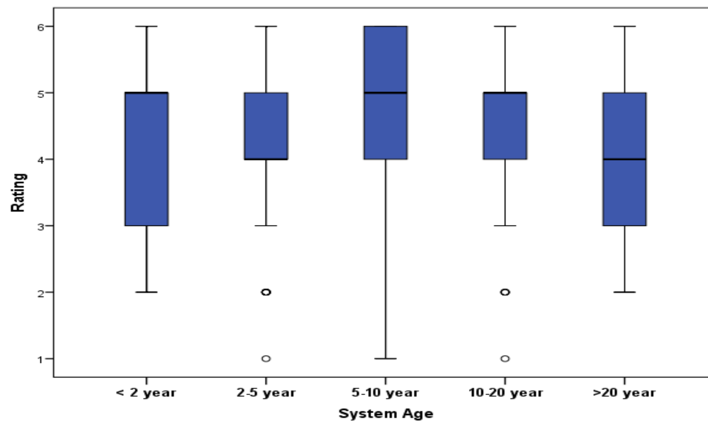


Figure 5. Distribution of understanding and/or measuring the issue

11.4.9. How professionals with different roles spend their extra time on different activities (RQ6)

It is inevitable that professionals having different roles will spend their extra time differently and on different activities. Accordingly, we wanted to explore how the roles estimate the extra time spent on the different activities. Fig. 8 compares the summary statistics on how the roles estimate the time spent on different activities, and it is evident from the figure that different roles spend their extra time on different activities.

We can see that Developers, Experts, Product Managers, and Project Managers spend most extra time on Understanding *and/or measuring the issues.* Meanwhile, Managers, Software Architects, and Testers spend most of their extra time on the *Management process.* Interestingly, the Managers represent the role that spends more extra time on the *Refactoring the issues,* and they are closely followed by Experts and Software Architects.

المنارة للاستشارات

www.r

Further, determined by one-way ANOVA, is a statistically significant difference between the mean value describing how the roles spend the time for the activities of *Finding the issue* (F(6)=2.201,p=.044), *Understanding and/or measuring the issues* (F(6)=2.894, p=.01), *Management process of the issues* (F(6)=4.336, p=.000).

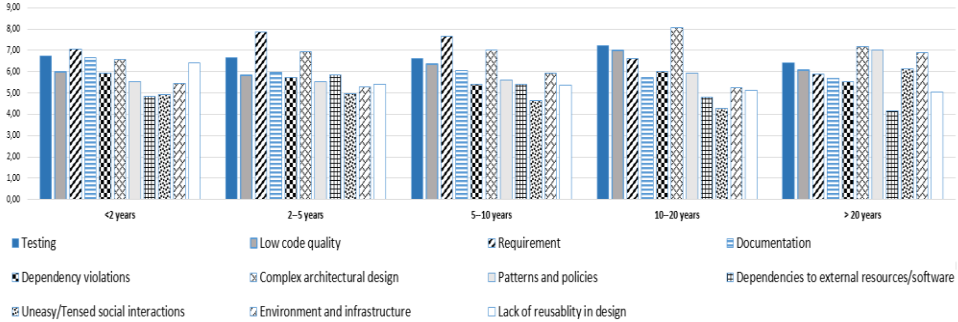


Figure 6. The mean value for each challenge in relation to system age

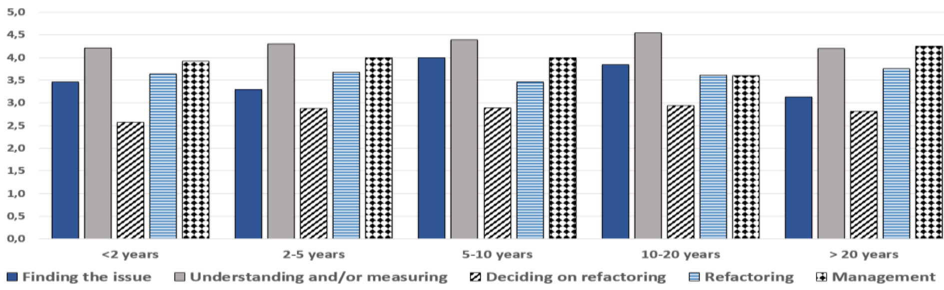


Figure 7. Extra time spent on different activities and system age

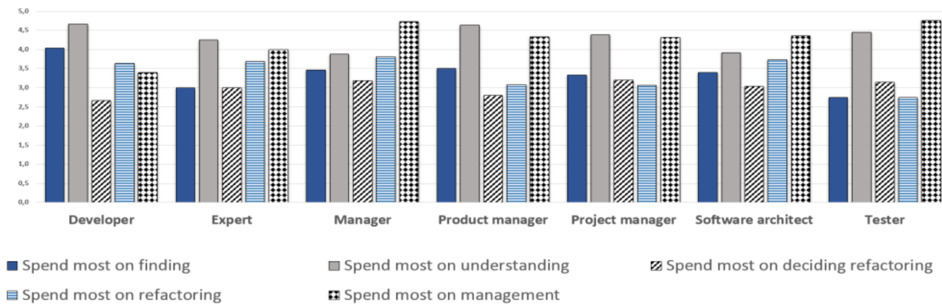


Figure 8. How professionals with different roles spend their extra time

المنارة للاستشارات

www.r

11.5. Discussion

11.5.1. Wasted time due to Technical Debt (RQ1,4,5)

The first research question (RQ1) aims to address how much of the overall development time is wasted because of TD. The striking results of the survey show that on average, the respondents estimate that **36%** of all software development time is wasted due to TD. The wasted time varies somewhat in relation to the system age but the distribution of the wasted time did not have a significant positive linear correlation with the system age (RQ4). However, the analysis from the survey shows that the most wasted time (40.5%) occurs in systems older than 20 years. Besides, even for new software systems with an age of less than two years, 33.8% of the time is estimated as wasted.

This result implies that TD is introduced early, and then it persists throughout the whole software life cycle.

The findings from the survey show that there is no significant difference in how the different roles estimate the overall wasted time (RQ6). However, the results show that Software Architects and Experts estimate the highest average value of wasted time (41%) and Developers estimate the lowest value of wasted time (35%). A possible explanation for these results may be that Software Architects have higher awareness about the complexity of the architectural design and, as identified in RQ2, a Complex Architectural Design is a common source of the most negative impact on daily software development work. Most of the interviewees explained that the magnitude of wasted time was consistent with their perception of the overall wasted time. They also claimed that this huge waste of time is detrimental to their business and its implications must be clearly addressed and recognized within their organizations.

11.5.2. What type of TD generates the most negative impact on daily development work (RQ2,4, and 5)

The second research question (RQ2) sought to determine what challenges generate the most negative impact on the daily software development work, in order to understand and weight different TD issues by potential severity. The overall survey results reveal that *Complex Architectural Design,* closely followed by *Requirement TD,* generates the most negative impact on daily software development work where the interviewees described the characteristics of a complex architectural design as *"One part of this is when you are afraid to change something because something which is completely unrelated gets corrupted through that. Then it is complex".* This finding broadly supports the work of other studies in this area by describing that TD instances linked to inadequacies in the software architecture are a major source of TD [17],[16], [149],[110]. Further, this study investigated the impact of the age of the software and if different roles experienced the *challenges* differently in the survey. According to these data, we can infer that the age of the system has a significant impact on *Architectural Complexity*, *Requirement TD and Social interactions* (RQ4). The impacts vary in relation to the system age (not linear), and *Complex Architecture Design* was observed having the highest negative impact on

المنارة للاستشارات
www.m

systems with an age interval of 10-20 years and the lowest value for the age interval of <2 years. The different role related findings report that Developers, Product and Project managers, and Testers rate the *Complex Architectural Design* as having the most negative impact (RQ5). Meanwhile, Software Architects interpret the *Requirement TD* to generate the most negative effect. These findings raise intriguing questions regarding if Software Architects are not prone enough to consider their own working area as the largest source of TD. However, another possible explanation for this could be that they actually have greater insight and understanding than others in this specific area. During the interviews, one of the interviewees commented on this result as "*Because software architects understand the architecture better and need requirements to create it, and thereby seeing the requirement as the biggest problem.*"

11.5.3. Extra time is spent on activities (RQ3,5, and 6)

The third research question (RQ3) focuses on how much extra time is spent on different activities due to the payment of TD interest. Both interviewees and survey respondents placed significant emphasis on describing that considerable time is spent on *Understanding and/or measuring* TD issues. This finding broadly supports the work of other studies in this area where the need for active TD management on especially architectural studies, is highlighted [97],[23],[150], [52]. Of interest is also the fact that *Understanding and/or measuring of Technical Debt* does not change due to the system age (RQ4), but is continuously at the highest level throughout the whole software life cycle. From the data, it is apparent that Developers, Experts, Product and Project Managers spend most extra time on *Understanding and/or measuring the issues.* Meanwhile, Managers, Software Architects, and Testers spend most of their extra time on the *Management process of the issues* (RQ6). This result could imply that different roles need different types of support.

11.6. Threats to Validity and Verifiability

For verifiability reasons, we have made information available online, to support a full or partial independent replication of the claimed contributions. All survey questions, used in this paper, are available on the link https://zenodo.org/record/437597#.WNOXw1PhCpo.

The qualitative data derived from the survey are not based on measured or observed data but on estimations made by the respondents. In future studies, we plan to include physical measurements and observations, to create a stronger reliability of the data. The result of this research may be affected by some threats to validity. *Construct validity* reflects what extent the operational measures that are studied represent, what the researchers have in mind and what is investigated according to the research questions [73]. To mitigate this risk and to make sure that the respondents were considering the correct type of TD issues, a short description of each type of TD was used and named as a *challenge*. Further, this study could possibly suffer from *internal validity* when causal relationships were examined as it affects our ability to explain the phenomena that we observed [66]. *External validity* focuses on to what extent it is possible to generalize the findings. There is always a risk in surveys that the sample is biased and for this topic, a potential threat

المنارة للاستشارات

www.m

refers to the demographic distribution of response samples. As reported in Section III.A.1, we mainly investigated companies from the Scandinavian area. To mitigate this validity issue, we attempted to enlarge the respondent's sample by inviting additional participants globally via LinkedIn. Without replicating this study to other countries, it is not possible to confirm that this study is generalizable. *Reliability* addresses whether the study would yield the same results if other researchers replicated it. To mitigate this threat, we have employed source triangulation, methodological triangulation and observer triangulation.

11.7. Conclusion

This is the first study surveying the estimated magnitude of the interest paid on the accumulated TD in terms of perceived wasted time and effort. This study is based on a survey with 258 respondents and group interviews with 32 practitioners. The study has shown that software development practitioners estimate that *36 % of all development time is wasted* due to TD and that *Complex Architectural Design and Requirement TD* generates the most negative impact on daily software development work. The most wasted time is spent on *Understanding and/or Measuring* TD.

This study reveals that all roles are heavily affected by the interest of TD, but different roles are affected differently. The study also shows that the age of the software affects the amount of wasted time and the different activities where the time is spent on. These findings have significant implications; organizations need to be aware of how much time and resources they are spending on their interest of TD and to deliberately focus on the remediation of their TD.

المنارة للاستشارات

www.r

www.n

12. Impact of Architectural Technical Debt on Software Development Work

The aim of this chapter is to investigate how practitioners perceive and estimate the impact of Architectural Technical Debt during the software development process.

The negative consequences of Technical Debt is an area of increasing interest, and more specifically the Architectural aspects of it have received increased attention in the last few years. Besides the negative effects of Architectural Technical Debt on the overall software product quality in terms of hindering evolution and causing high maintenance costs, Architectural Technical Debt also has a significant negative impact on software practitioners' daily work. Although a great deal of theoretical work on Architectural Technical Debt has been undertaken, there is a lack of empirical studies that examine the negative effects of Architectural Technical Debt during the software development lifecycle. This paper reports the results of an online web survey providing quantitative data from 258 participants. The contribution of this paper is threefold: First, it shows that practitioners experience that the Architectural type of Technical Debt has the highest negative impact on daily software development work. Secondly, we provide evidence that does not support the commonly held belief that Architectural Technical Debt increases with the age of the software. Thirdly, we show that despite different responsibilities and working tasks of software professionals, Architectural Technical Debt negatively affects all roles without any significant difference between the roles.

This chapter has been published as:
Impact of Architectural Technical Debt on Daily Software Development Work – A Survey of Software Practitioners
T. Besker, A. Martini, and J. Bosch
In Proceedings of the 43th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017.

12.1. Introduction

Technical Debt (TD) is recognized as a critical issue in today's software development industry [7]. Since this phenomenon has a negative effect on software companies, it is important to assess and estimate the consequences of TD, both in terms of the wasted time it causes and its negative effects on daily software development work.

Ward Cunningham [91] introduced the financial metaphor of TD to describe to nontechnical product stakeholders about the need to identify the potential long-term and far-reaching negative effects of the immature code that is implemented during the software lifecycle.

A more recent explanation was provided by Avgeriou et al. [4] who describe TD as "*In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can*

المنارة للاستشارات
www.r

make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." The reason for taking on TD can be defined as an unintentional consequence resulting from the accumulation of non-optimal decisions over time or, as in Cunningham's definition, intentionally in order to get new functionality running quickly [151].

Interest is the negative effects of the extra effort that has to be paid due to the accumulated amount of TD in the software, such as executing manual processes that could potentially be automated or expending excessive effort on modifying unnecessarily complex code, performance problems due to lower resource usage by inefficient code, and similar costs [7],[151]. The interest can also be described as "*The additional effort that is needed to be spent on maintaining the software because of its decayed design-time quality*" [30].

Today, it has been well established from various studies that TD has a negative impact on software development organizations by impeding evolution and causing high maintenance costs due to internal software quality issues [18], [109], [22], [24], [103]. Left unmanaged, TD can result in significant cost overruns, severe quality issues, a limited ability to add new features [50] and even result in reaching a crisis point when a huge, costly refactoring or an entire replacement of the system needs to be undertaken [5].

During development of large-scale software systems, the software architecture plays a significant important role [18] and consequently a vital part of the overall TD relates to sub-optimal architectural decisions and is regarded as *Architectural* Technical Debt (ATD) [23].

ATD is primarily incurred by architectural compromises with the consequence of immature architectural artifacts such as violations of best practices [93], consistency and integrity constraints of the architectures, or implementation of immature architectural techniques. These concerns chiefly result from the compromise of performance efficiency, reliability, maintainability, reusability, and the ability to add new features [95].

Besides the negative effects ATD has on the overall software product quality, ATD also has a significant negative impact on software practitioners' daily work. It is, therefore, important to understand *when*, and to *whom,* and in *what way* (in terms of generating wasted time) ATD has a negative impact during the software development process. In this study, the negative effects of ATD from a software lifecycle perspective are examined by conducting a web survey of 258 industry practitioners. This paper has four key goals:

Firstly, there are no studies quantifying the interest in terms of how much (observed, measured or estimated) time is wasted due to *Architectural* TD. We aim to understand the level of negative effects ATD has on the daily software development work and compare this level with negative effects due to other types of TD.

Secondly, we aim to understand if the level of the negative effects due to ATD correlates with the estimated wasted development time during the software lifecycle.

Thirdly, the negative effect due to ATD is commonly believed to have an increasingly negative impact with respect to the age of the software. This study aims to assess the

المنارة للاستشارات

www.r

extent to which the level of negative effects due to ATD, differs in relation to the age of the system.

Finally, during the software lifecycle, several different professional roles participate, and could subsequently be affected differently by ATD. To the best of our knowledge, no previous studies have examined how different software professional roles perceive the negative effects specifically generated by ATD. Therefore, this paper analyses the variation in the negative impact of ATD on different software roles on their daily software development work.

With regards to these four goals, we aim to answer the following research questions (RQ):

RQ1. *Does ATD generate a negative impact on daily software development work?*

RQ2. *Does ATD negatively affect the amount of wasted development time?*

RQ3. *Does the Age of the software system affect the negative effects due to ATD?*

RQ4. *In what way are different software Roles affected by ATD?*

The research questions will be answered using estimated survey data based on software professionals' perceptions. All survey respondents were experienced in software development, and therefore, their estimates were likely to be formed by what they have heard, observed, and experienced at their companies.

This study makes an original contribution to research, with respect to the existing body of knowledge relating to ATD in the following important areas:

1. We provide an empirically based study on how practitioners estimate and experience ATD within the software industry, based on empirical quantitative data.
2. We present results that confirm that ATD has a high negative impact on the practitioners' daily software development work.
3. This estimated wasted time by the practitioners does not correlate with the level of negative impact generated by ATD, on daily software development work.
4. This study shows that the level of negative impact due to ATD is introduced early, and thereafter remains during the whole software lifecycle. Based on evidence from our survey, this study does not support the currently held belief that the negative effects due to ATD increase with respect to the age of the system.
5. This study provides new insights into ATD research by showing that ATD has an extensive negative effect on all software professional roles.
6. This study demonstrates to both practitioners and academics the importance of paying more attention and effort to early remediate ATD during the software lifecycle, in order to decrease the level of negative impact due to ATD on daily software development work.

The remainder of this paper is structured in seven sections. The next section introduces related work. In the third section, the research method is described. The fourth section presents the analysis, and the results that are discussed in section five. Finally, in section six, threats to validity are presented, while section seven concludes the paper.

المنارة للاستشارات

www.n

12.2. Related work

Research addressing the negative effects and impacts of TD in general is relatively well represented within academia [150], but there is currently no academic research quantifying how much software development time is estimated to be *wasted* due to *Architectural* TD and if such wasted time varies in relation to the level of the experienced and estimated negative effects generated by ATD. A previous study by Li et al. [96] states that ATD concerns different types of roles in different ways; however, this current study fills a gap in the literature by addressing to what extent different professional roles perceive the negative effects of ATD and exploring how these negative effects vary during the software lifecycle, in terms of the system age.

12.2.1. The importance of addressing ATD

ATD is an important component that needs to be addressed in the architecting process [96], in order to make sure that the advantages of sub-optimal solutions do not lead to large future payments of interest [127], [121], [32].

The importance of ATD has been highlighted in several studies with statements such as, “the most encountered instances of technical debt are caused by architectural inadequacies” by [17], “we found that architectural decisions are the most important source of technical debt” by [16], and “because architecture has such leverage within the overall development lifecycle, strategic management of architectural debt is of primary importance” [28]. In our previous paper [5], we pointed out the risks when ATD grows until the result of negative effect causes adding new business value so slowly that it becomes necessary to conduct extensive refactoring or even rebuilding the complete software from scratch.

12.2.2. Survey-based studies on TD

By reviewing other survey-based studies focusing on TD, we, in particular, found four studies which bear a resemblance to this study.

First, Ernst et al. [16] conducted a survey within three large organizations, with 536 respondents (primarily software engineers and architects). This research included follow-up interviews with seven software engineers. Similar to this study, one of their main research areas focused on how much of the TD is architectural in nature. Their result shows that “bad architecture choices” are the greatest source of TD. The study also determines that the degree of architectural drift is related to system age. They found a weak association between the age of the system and the perceived importance of architectural issues where 89% of the respondents with software age ≥6 years agreed or strongly agreed with the notion that architectural issues were a significant source of debt, compared to 80% of those with newer systems (< 3 years). Compared to this study, our study has a higher granularity of the age intervals and therefore provides a more detailed description from a lifecycle perspective. In contrast to that study, our study also includes the estimated wasted time due to ATD.

المنارة للاستشارات

www.r

Secondly, [17] conducted a web survey of Finnish software practitioners to determine their level of TD knowledge, how TD manifests in their projects, and which of the applied components of agile software development are sensitive to TD. Consequently, they found that the most frequently indicated causes of TD were inadequate architecture and inadequate documentation. This study was conducted in Finland and included 54 applicable responses. However, in these studies, we cannot find a quantification of the interest, and furthermore, there are no examinations on the consequences ATD specifically has for different software roles.

This paper is, to some extent, also related to our previous papers [144],[95], where we study and compare several different TD types and address how TD negatively affects different software quality attributes. However, even if the data are collected using the same survey, this study focuses on the Architectural aspects of TD, and does not include any effects on the software quality attributes and does not focus on other types of TD. By only focusing specifically on ATD, this means that we can provide a more in-depth analysis of the architecturally related issues of TD and provide more detailed statistical analysis of the data.

12.2.3. Research on TD interest variations

Kazman et al. [116] have published a case study identifying and quantifying architectural debts in an industrial software project, using code changes as a proxy for calculating the interest. This study focuses on identifying the architectural roots of TD, meaning that the study does not report the cost of interest, but instead targets quantifying the expected payback for refactoring. These costs are, to some extent, based on estimated values from the interviewed architects and, based on these assumptions, the expected benefit from the refactoring is calculated. By comparison, this study has a focus on quantifying the interest in terms of wasted time instead of on locating the roots of the architectural TD and the expected benefit from the refactoring.

Falessi et al. [104] argue that the interest of TD can change over time, is non-linear and has a probability of growing exponentially rather than linearly. By examining source code, Nugroho et al. [142] state that the interest grows differently on a 10-year horizon, depending on a star rating system for the software.

Xiao et al. [117] proposed an approach to identifying and quantifying ATD, focusing on modeling the growth and on the quantification of the maintenance cost. In this study, the interest is calculated using approximations based on the number of lines of code modified and committed to fix bugs. This work has the limitation of only including the cost of maintenance as ATD and primarily focuses on bug fixes.

Although some research has been conducted on addressing the growth of the interest, there is a lack of empirical investigations quantifying the interest from an architectural perspective in terms of wasted time.

المنارة للاستشارات

www.r

12.3. Methodology

The following sections describe the methods used for conducting the survey, the data collection, and the data analysis.

12.3.1. Design of the Survey

The web survey was designed and hosted by the online survey service Survey Monkey. The questions were a combination of optional and mandatory. To avoid bias in the survey, the questions were developed as neutrally as possible, in such a way that one question did not influence the response to the next, and clear, unbiased instruction was provided when needed [78].

The survey was divided into two different sections in order to give the respondents a context and understanding for each set of questions. The first draft of the survey was tested by four industrial practitioners (developer, manager, project owner and software architect) and by two Ph.D. students in order to evaluate the understanding of the questions and the usage of common terms and expressions [77]. During this evaluation, we also monitored the time that was needed. The survey was made accessible between February and March 2016, and a reminder was sent out after two weeks to those who had been specifically invited.

The survey was anonymous, and participation in the survey was voluntary. All invited respondents were informed about the survey with an invitation text describing the motives and goals of the study.

12.3.2. Data collection of survey data

The invitation to the survey was emailed directly to seven companies/partners within our networks (located in Scandinavia, with an extensive range of software development), and invitations were also distributed at software engineering-related networks on LinkedIn. Across all these collaborators, 312 respondents began the survey, and 258 respondents answered all questions. Due to this high completion rate (83%), the decision was made to reject the unfinished questionnaires, according to guidelines by [78].

The first part of the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their companies. This data is compiled and presented in Table I. The level of education of the respondents was relatively high, with 58% having a master's degree and 25% having a bachelor's degree. The survey included respondents having different professional roles, where 49% were Developers/Programmers/Software Engineers (abbreviated to Developers in this paper), while 25% were Software Architects. Approximately 78% of the Software architects and 59% of the Developers-/Programmers/Software Engineers had more than 10 years of experience. The most common size of the software development team included 6-10 members (36%), and the majority (32%) systems were on average 5-10 years old from the initial design. Nevertheless, a significant number (35%) of the respondents' software was more than 10 years old. As indicated in Table I, the majority of the software systems were embedded

المنارة للاستشارات

www.r

systems (49%) and real-time systems (42%). However, for this specific question, the respondents could select more than one option.

TABLE I - CHARACTERISTICS OF THE SAMPLE SURVEY

Individual data		Company data	
Experience		**Team size**	
< 2 years	3,90%	1-5 members	23,30%
2 - 5 year	10,50%	6-10 members	36,00%
5 - 10 year	17,40%	11-20 members	15,90%
> 10 years	68,20%	21-40 members	6,60%
		> 40 members	18,20%
Education		**Software system type***	
Master's degree	57,80%	Embedded system	48.84%
Bachelor's degree	25,20%	Real-time system	41.86%
No Univ. education	6,20%	Data management system	22.09%
Other:	5,80%	System Integration	20.93%
Ph.D. degree	5,00%	Modeling and/or simul.	15.12%
		Data analysis system	14.73%
Roles		Web 2.0 / SaaS system	8.53%
Developer/Program/-Software Engineer **	49,20%	Other	8.53%
Software Architect	24,80%	**System Age**	
Manager	6,20%	< 2 years	9,70%
Project Manager	6,20%	2-5 years	23,3%
Product Manager	5,0%	5-10 years	32,2%
Expert	5,0%	10-20 years	28,3%
Tester	3,50%	>20 years	6,6%
Gender			
Male	89,90%		
Female	8,50%		
Other/no share	1,60%		

* More than one option was selectable, ** Abbreviated as ***Developer*** in this paper

The second part of the survey focused on questions based on the research questions presented in Section I. In this part of the survey, the following survey questions (SQ) were asked:

SQ1. Which of the following challenges generates the most negative impact on your daily software development work? Please rank them from 1 to 11.

SQ2. How much of the overall development time is wasted because of these issues? (Do not consider time spent fixing and managing the issues, just if they hinder you when you add/change/understand the system).

For the first question SQ1, the listed categories and sub-categories (Complex Architectural Design, Requirement TD, Testing TD, Source Code TD, Documentation TD, Too many different patterns and policies, Dependency violations, Infrastructure TD, Lack of reusability in design, Dependencies to external resources/software, Uneasy/Tensed social interactions between different stakeholders) provided by [15] were used to distinguish between different TD types. All these TD types were referred to as "Challenges", in order to provide the respondents with a more detailed clarification of the TD type. This also mitigated the risk of misinterpretations and helped to make sure that the respondents were considering the correct type of TD issue.

المنارة للاستشارات

www.r

In order to ensure a better construct validity of the survey, these same types of TD were referred to in survey question SQ2. In SQ2, the estimation of the interest as wasted time was specified in predefined percentage intervals of < 10%, 10-20%...80-90% and I don't know. In order to separate the time spent on management, the respondents were asked not to include the time spent on fixing and managing the TD issues.

12.3.2.1. Data analysis of survey data

The data from the survey were analyzed using a quantitative method, by interpreting the numbers obtained from the answers to the survey. All statistical analyses were carried out using the software SPSS (version 22). Descriptive statistics were employed to organize and analyze the qualitative data by using tables and charts.

The data were analyzed by studying the median, mean and standard deviation and also by using the statistical methods Pearson's R, the Pearson chi-square tests and ANOVA.

The Pearson's R method computes pairwise, determining the strength and direction of the association between two metrics and can be used to measure a linear association between two metrics.

The Pearson chi-squared tests are used for evaluating how likely it is that any observed difference between the sets arose by chance, and the test of independence assesses whether unpaired observations on two variables are independent of each other.

The one-way ANOVA was used to identify significant differences in mean values.

12.4. Results and Analysis

The following subsections present the analysis and results for the research question presented in Section I, and the results are presented according to each research question.

12.4.1. Does ATD generate a negative impact on daily software development work? (RQ1)

In order to understand the level of negative impact ATD has on the practitioners' daily software development work, this TD type was compared with several other TD types. The survey respondents were asked to rank (score 1 = lowest negative impact, score 11 = highest impact) a randomly ordered list of different TD types they consider to have the most negative impact on their daily software development work. Each of these challenges corresponds to different TD types which reflected the TD types that emerged from [15]. For example, if one of the TD types was ranked as generating the *most* negative effect by one respondent, it was given a score of 11, and if second 10 and so on, and the score 1 was given if the negative impact was ranked as generating the *least* negative effect.

المنارة للاستشارات

www.r

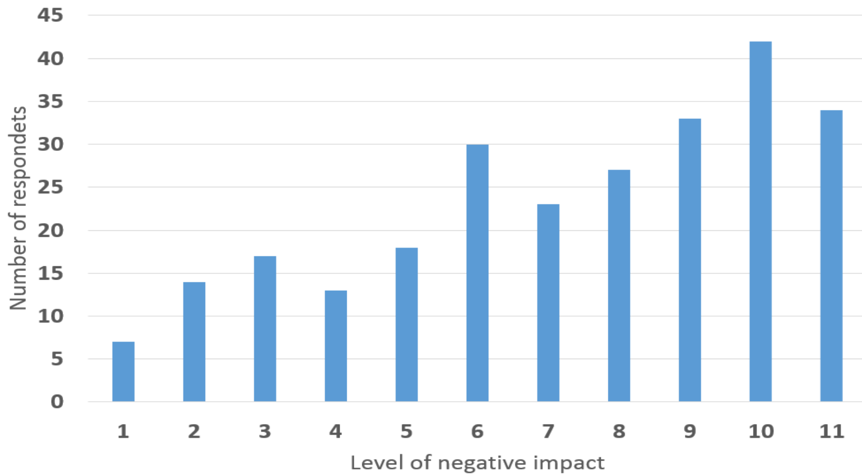


Figure 1 Distribution of the negative effects due to ATD

The results of the level of negative effects due to ATD in terms of Complex Architectural Design (CAD) are presented in Fig 1. From the data in this figure, it is apparent that the number of respondents is increasing in relation to the perceived negative impact of ATD. A total of 109 (42%) of all the respondents experience the three highest levels of negative effect (score 9, 10 and 11) due to ATD. Only seven (2.7%) respondents perceive that ATD has the least negative impact, while 34 (13.2%) respondents estimate ATD has the most negative impact on their daily work.

In our previous study [144], we studied how several different types of TD have an negative effect on the practitioners' daily software development work. To understand the impact of ATD, we have, in this study, compared the negative effects due to ATD with data from the previous study.

Table 2 presents the summary statistics for each of the listed challenges in the survey, based on the results of the other TD types presented in our previous study [144]. The mean value for each TD type, is the average of the weights associated with the ranking reported by the respondents in the survey.

Given these weights, a mean value was calculated for each TD type: this way, we could see which challenges were ranked as having the most and least negative impact on the daily software development work. Table 2 reveals that CAD (mean rank 7.27 with a standard deviation of 2.9) generates the most negative impact on daily software development work, both when studying the mean and the median values. The level of negative impact due to a complex architectural design is closely followed by Requirement Technical Debt (mean rank 7.23).

المنارة للاستشارات

www.r

TABLE 2 - MEAN AND MEDIAN OF THE RANKINGS FOR TD TYPES

Challenge	Mean	Median
Complex Architectural Design	7,27	8
Requirement TD	7,23	8
Testing TD	6,80	7
Code TD	6,36	7
Documentation TD	5,98	5,5
Many patterns and policies	5,76	5
Dependency Violations	5,71	6
Infrastructure TD	5,59	5
Lack of reusability	5,38	5
Dependencies to externals	5,19	4
Social TD	4,73	4

12.4.2. Does ATD negatively affect the amount of estimated wasted development time? (RQ2).

This research question focuses on the interest in terms of how much of the overall software development time is wasted due to paying the interest of ATD. In our previous paper [144], we reported that the respondents estimate that 36% of all software development time is wasted because of TD in general, with the standard deviation of 17.8%.

In this specific research question, we seek to confirm the commonly held belief in a correlation between a higher level of negative effect and a higher amount of wasted development time.

To evaluate if there is such a significant correlation between the estimated overall wasted development time and the experienced level of negative impact due to ATD, we have correlated these two datasets.

When testing the belief that the estimated wasted time increases in relation to the level of negative impact generated by a complex architectural design, in terms of ATD, we calculated the average estimated wasted time within each of the levels of the negative impact (the same ranking scale as in Section IV.A, were used where level 1 = lowest negative impact and level 11 = highest negative impact due to ATD).

المنارة للاستشارات

www.r

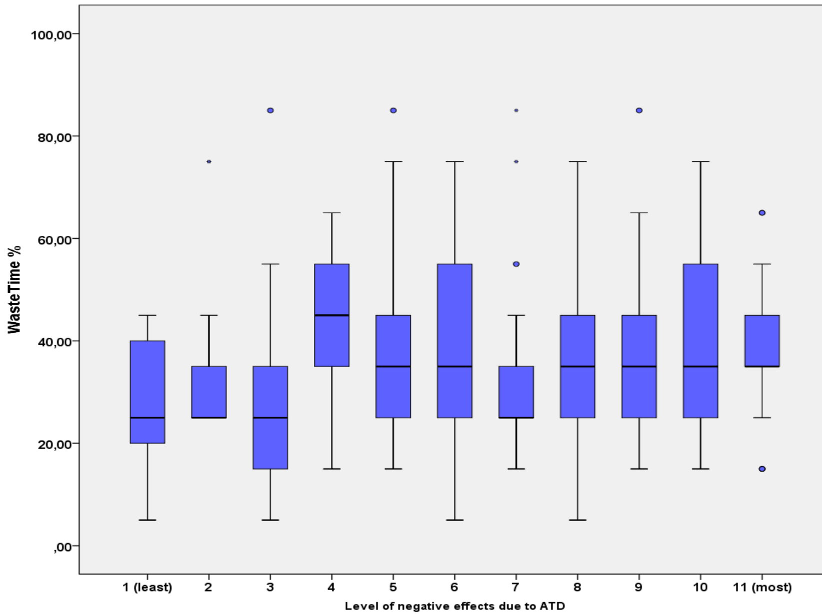


Figure 2. Estimated wasted software development time for each level of negative effects due to ATD

Fig. 2 represents a boxplot for each level of the negative effect due to ATD. In this figure, the estimated wasted time is plotted and compared using the different levels of negative impacts on the daily software development work due to ATD.

This boxplot represents the minimum and maximum range values, the upper and lower quartiles, and the median for each level. From the boxplot, we can see that the estimated wasted time is relatively consistent within the different levels of negative impact. The maximum estimated wasted time (41%) was found within level 4 of the negative impact and the minimum wasted time (30%) was found in the first and the third level of negative effects due to ATD.

In Fig. 3, the differences of distributions of the estimated wasted time in relation to the mean value of the negative effect generated by ATD are graphically visualized. Respondents who estimate they are wasting less than 10% of their working time due to TD, estimate the lowest negative effect due to ATD (mean value 4.71) and respondents estimate they are wasting the most wasted time (80-90%) estimate the second lowest negative effect (mean value 6.0).

المنارة للاستشارات

www.r

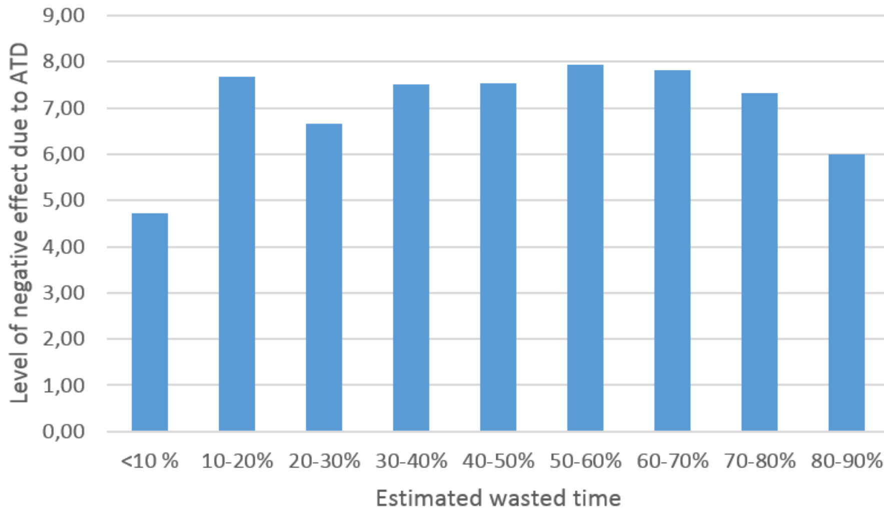


Figure 3. Mean value of the negative effect due to ATD in relation to the estimated wasted time

The analysis determined by one-way ANOVA showed that the estimated wasted time was not statistically significantly different between the levels of impact.

Furthermore, to evaluate if there is a linear correlation between the level of the negative impact caused by ATD and the wasted time, we used the Pearson correlation method. These two data series (we used the average wasted time within each interval), returned p-value 0,084, which showed no support for a linear correlation between the estimated wasted time and level of negative impact due to ATD.

Overall, this analysis did not confirm a statistically significant correlation between the estimated wasted time and the level of experienced negative impact ATD has on the respondents' daily software development work. '

This result indicates that the respondents who stated that ATD generates the most negative effects (rank 11) on their daily work, did not waste significantly more or less amount of time compared to the respondents stating that ATD generates less negative effects.

In summary, the perceived negative effects due to ATD did not affect the respondents' estimated wasted time and, based on evidence from our survey, this study does not support the currently held belief that the negative effects due to ATD significantly correlates or linearly increases with respect to the age of the system.

12.4.3. Does the Age of the software system affect the level of negative effects due to ATD? (RQ3)

There is a commonly held belief that the negative effects of a complex architectural design, in terms of ATD, increase with the age of the software, commonly referred to as software aging [37].

المنارة للاستشارات

www.r

In order to assess whether the negative effects of ATD varies during the software lifecycle, we used a statistical cross-tabulated analysis showing the negative effects caused by ATD (using the same ranking scale as in Section IV.A) on the daily software work in relation to different software age intervals of < 2 years, 2-5 years, 5-10 years, 10-20 years, and > 20 years.

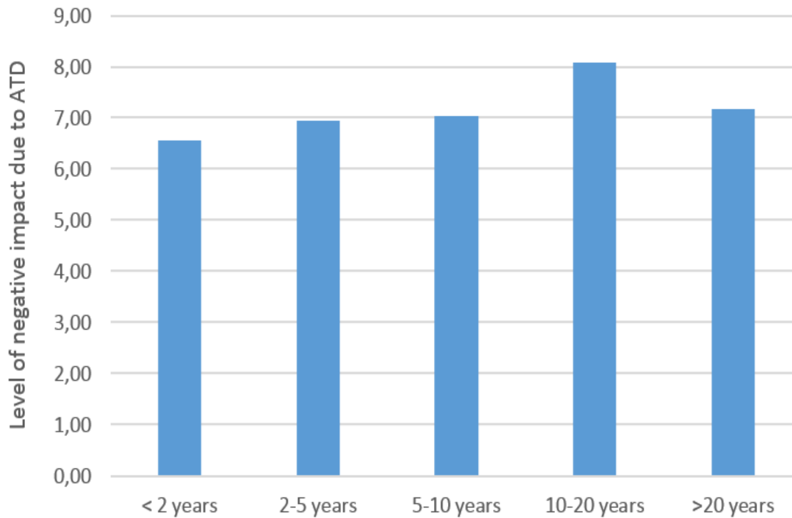


Figure 4. Distribution of negative effect (mean values) for each software age interval

To calculate the mean value for the ages of the software, the average of the reported age interval stated by the respondents in the survey was used. For example, if the age was reported within the interval of < 2 years, the average age was set to 1 year, and for the interval 5-10 years, the average age of 7.5 years was used. For systems reported > 20 years the age of 25 was used.

The first set of analyses examined the negative effects due to ATD for each software age interval and, as shown in Fig. 4, the mean values of the negative effect somewhat vary over the different age intervals. What is interesting about the data in this figure is that, for software which is less than two years, the negative impact of ATD has already a relatively high negative impact on the respondents' daily software work, with a mean value of 6.56. This result implies that ATD is introduced quite early during the software lifecycle and is also generating extensive negative effects for systems with ages less than two years. After the first two years, the negative effects have a growing trend up to the peak (mean value 8.08) in the age interval of systems with ages between 10 and 20 years. For systems older than 20 years, the negative impact somewhat decreases (mean value 7.18).

To further evaluate if there is a linear correlation between the system age and level of negative impact, we used the Pearson correlation method. Despite the visual impression in Fig. 4 somewhat indicating a small increase in the software age of the system, our statistical analysis of the two data series (we used the average age within each year interval) did not return any indication of a significant linear correlation between the system age and the level of negative effects due to ATD (r(256)=,127, p=0,041). The boxplot in Fig. 5 captures the difference in software age in relation to each level of

المنارة للاستشارات
www.r

negative impact due to ATD. Looking at the comparison of median and percentiles among the different levels of negative impact, we cannot see a significant difference with regards to the age distribution, besides for level 10 (median 15 years) and level 5 (median 5.5 years). The mean values of the system age vary between a minimum of 6.1 years (for level 5) and a maximum of 12.1 years (for level 10) among all the different levels. To evaluate if the software age is significantly different among the different intervals, a one-way ANOVA test was used. This test revealed that the age of the system is significantly different among the different levels of impact (F(10,247)=1,928, p=0.042). Furthermore, the Pearson chi-square test of dependence showed that the two factors (system age and level of negative impact) are not dependent, which means that there is no significant relationship between the age of the system and the experienced level of negative effects due to ATD. Hence, we did not find any statistically verifiable evidence showing a correlation between the age of the software and the level of negative effects generated due to ATD on the respondents' daily work. The result shows that, for young software systems with an age of less than two years, the negative impact generated by ATD is also quite extensive. Based on evidence from our survey, this study does not support the currently held belief that the negative effects due to ATD, significant correlates or linearly increases with respect to the age of the system.

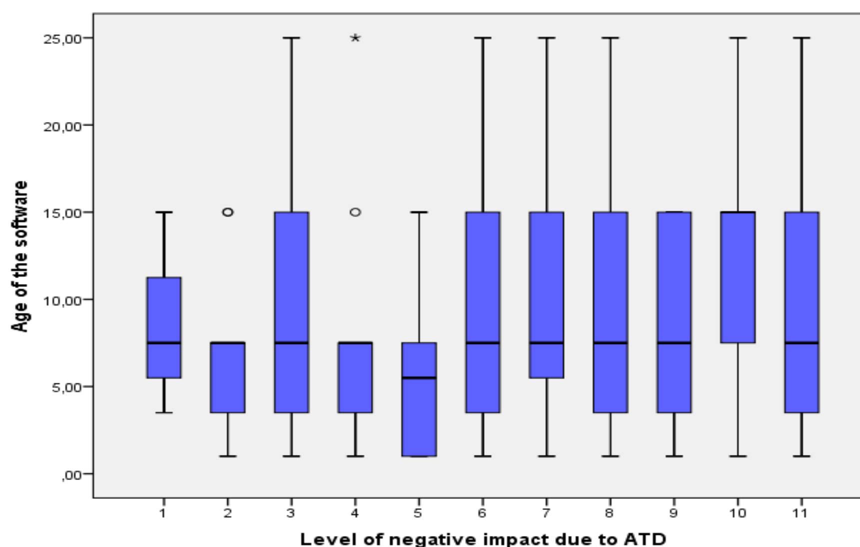


Figure 5. Age of the software for each level of negative effects due to ATD

12.4.4. ATD impacts on different Roles (RQ4)

In our previous study [144], we found that, among all different TD types, ATD has the greatest negative impact on the daily software development work for all the different roles. In this section, we explore if the different professional software roles (Developer, Software Architect, Manager, Project Manager, Product Manager, Expert, and Tester) experience the negative effects generated by ATD, on their daily software development work differently. This research question uses the same ranking scale as in Section IV.A,

المنارة للاستشارات

www.r

where level 1 represents the lowest negative impact and level 11 represents the highest negative impact generated by ATD. By examining the mean values in Fig. 6, it is evident that Project Managers (mean value 8.31) and Testers (mean value 8.11), estimate the highest negative impact due to ATD whereas Experts (mean value 6.46) and Software Architects estimate the lowest value of the negative impact of ATD (mean value 6.67).

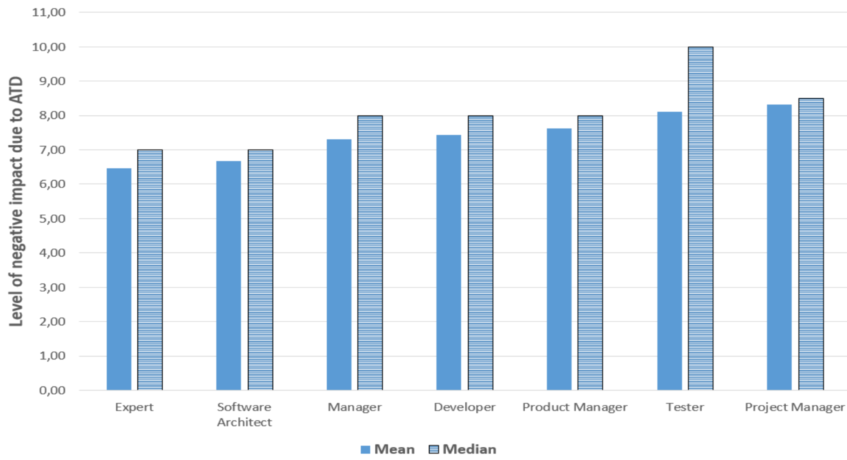


Figure 6. Different roles' level of negative effects due to ATD

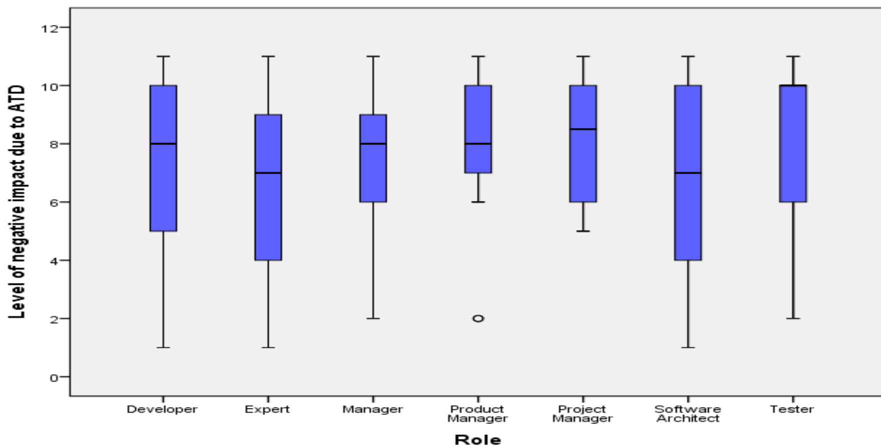


Figure 7. Distribution of different roles' level of negative effects due to ATD

To graphically visualize the differences of distributions for each negative impact level, Fig. 7 presents a boxplot for each of the roles. From this boxplot, we can see the variation and distribution of how the roles estimate the negative impact due to ATD.

When explicitly tested if the roles experience the negative effects due to ATD differently, a one-way ANOVA test showed, somewhat surprisingly, that there were no significant differences ($F(6,251)=1,16, p=0.310$) between how the roles are negatively affected by ATD.

المنارة للاستشارات

www.r

One unanticipated finding was that five of the Software Architects (8%) stated that ATD generates the least (level 1) negative impact on their daily work. Meanwhile, 19% of the Project Managers perceive that ATD has the highest level of impact on their daily software development work.

The result of this research question indicates that *all* different software professional roles are extensively negatively affected by ATD.

12.5. Discussion and limitations

This section presents and discusses the findings and the implications of the research questions presented in Section I.

12.5.1. The negative impact on daily software development work due to ATD (RQ1)

Several previous studies have described that ATD has a severe negative impact on software development in general, but there is, to the best of our knowledge, no research specifically focusing on quantifying the level of negative impact ATD has on practitioners' daily software work.

The first research question (RQ1) aims to address the level of the negative impact ATD has on the respondents' daily software development work. The result shows that ATD is ranked as having a high negative impact on the practitioners' daily software development work. Moreover, compared to other types of TD [144], this survey reveals that *Complex Architectural Design,* closely followed by *Requirement TD,* generates the most negative impact on daily software development work. This finding broadly reflects the findings from other studies in this area by describing that TD instances linked to inadequacies in the software architecture are major sources of TD [17], [16], [149], [110].

12.5.2. Does ATD negatively affect the amount of wasted development time? (RQ2).

The second research question (RQ2) focuses on how much of the overall software development time is estimated to be wasted because of ATD and also to evaluate the commonly held belief of a correlation between this estimated wasted development time and the experienced level of negative impact generated by ATD.

A common belief exists that there is a correlation between the estimated wasted software development time and the negative effects due to ATD, but this study provides empirical evidence which does not support such belief. The level of negative effects generated by ATD thus does not have a significant correlation between the amount of estimated wasted software development time. In contrast, the estimated wasted time did not differ significantly among the different levels of negative effects generated by ATD. However, the estimated wasted time varies in relation to the level of negative effects due to ATD, but the distribution of the wasted time did not have a significant increase (or decrease) linear correlation with respect to the different levels of negative impacts.

المنارة للاستشارات

www.r

Contrary to popular belief, how negatively respondents estimate the impact due to ATD proves to be a poor indicator of wasted software development time. Therefore, studies asking software professionals how often they encounter ATD, how they perceive the importance of ATD, or how they experience the negative impact due to ATD, should be carefully examined. This is because our research shows that there is no correlation between the level of negative effects due to ATD and the amount of the estimated wasted time.

Further research should be undertaken to investigate if other types of TD (besides ATD) have a significant correlation to the estimated wasted time to better understand the negative impact different TD types have on the wasted software development time.

12.5.3. Does the Age of the software system affect the negative effects due to ATD? (RQ3)

The third research question (RQ3) in this study sought to determine if there is a correlation between the level of negative impact generated by ATD and the age of the software.

It is a commonly held belief that the negative effects generated by ATD increase over time within the software lifecycle. However, the findings of this study do not support such a belief. The negative effects due to ATD vary in relation to the system age (but not linear), and were observed as having the highest negative impact on software with an age interval of 10-20 years and the lowest value for the age interval of < 2 years.

What is interesting about the result is that for young software with an age of less than two years, the negative impact generated by ATD is also quite extensive. A possible explanation for these results may be that ATD is introduced early in the software, and then persists throughout the whole software lifecycle. This result implies that investment in refactoring must be continuous from the conception of the system in order to keep the negative effect generated by ATD at a future low level.

12.5.4. ATD impact on different Roles (RQ4)

The fourth research question (RQ4) explores the negative effects ATD has on different types of professional software roles. Both technically oriented professionals such as developers, testers, and experts and also professionals working within a management area of the software, such as software architects, product and project managers, participate in the software development process during the software lifecycle. All these roles have different responsibilities and tasks and could subsequently potentially be differently affected by ATD. The statistical analysis could not confirm the negative impact generated by ATD among the roles as significantly different. This study confirms that ATD has an extensive negative effect on *all* software professional roles within the companies.

12.5.5. Verifiability and Limitations

For *verifiability* reasons, we have made information available online to support a full or partial independent replication of the claimed contributions. All survey questions used in

المنارة للاستشارات

www.r

this paper are available on the link https://zenodo.org/record/230742#.WG34rP7rupp. The reported findings are subject to at least two limitations. First, the majority of the data from the invited companies for the survey and the interviewed companies were gathered in Scandinavia. Thus, the results might be different in other geographical and cultural areas. As a result, further work is needed to replicate the results in other geographical areas and software development cultures. Second, the qualitative data derived from the survey are not based on measured or observed data but rather on estimations made by the respondents. Even if there was a high degree of agreement across the respondents' estimations, this might create bias for the results in terms of their credibility and correctness. Assessment of TD is, however, not an exact science; it all depends on how you calculate it [152]. Therefore, as a future study, we plan to investigate this area further, to also include physical measurements and observations, in order to create a stronger reliability of the data.

12.6. Threats to validity and Verifiability

The result of this research may be affected by some threats to validity, such as construct validity, internal validity, external validity, and reliability.

Construct validity reflects to what extent the operational measures that are studied represent what the researchers have in mind, and what is investigated according to the research questions [73]. To mitigate this risk and to make sure that the respondents were considering the correct type of TD issues, a short description of each type of TD was used and named as Challenge.

This study could potentially suffer from *internal validity* when the causal relationships were examined, as it affects our ability to accurately explain the phenomena that we observed [66].

External validity focuses on to what extent it is possible to generalize the findings. There is always a risk in surveys that the sample is biased and, for this topic, a potential threat refers to the demographic distribution of response samples. As reported in Section III.A.1, we primarily investigated companies from the Scandinavian area. To mitigate this validity issue, we attempted to enlarge the respondents' sample by inviting additional participants globally via LinkedIn. Without replicating this study to other countries or regions, it is not possible to confirm that this study is generalizable.

Reliability addresses whether the study would yield the same results if other researchers replicated it. To mitigate this threat, we have employed observer triangulation, which means that all authors participated in the analysis.

12.7. Conclusion

Architectural Technical Debt is evidently detrimental to software companies, and it is important to assess and estimate its negative consequences on daily software development work. The main goal of this study was to determine how practitioners within the software industry perceive and estimate the negative effects due to ATD in terms of the level of negative effects.

المنارة للاستشارات

www.r

To the best of our knowledge, this is the first study that empirically surveys the estimated detrimental impact of ATD in terms of estimated wasted time and effort. This study is based on a survey with 258 respondents. The study has shown that ATD has a high negative impact on the practitioner's daily software development work. The research has also shown that the amount of wasted time does not have a statistically significant linear correlation with how the practitioners experience the level of negative effects due to ATD. Practitioners waste time statistically independently of how they experience the negative impact of ATD.

Furthermore, this study reveals that ATD has an extensive negative effect on *all* software professional roles and, additionally, our study provides evidence which does not appear to support the commonly held belief that the ATD increases with the age of the software, as this study did not find any statistically verifiable relationship between the age of the software and the level of negative effects generated by ATD. The result shows that ATD is introduced early and persists during the whole software lifecycle.

These findings have significant implications, advocating for the important awareness of how much valuable time and effort is wasted due to ATD. These findings provide strong empirical confirmation that software companies need to invest in continuous refactoring from the conception of the system in order to keep the negative effect generated by ATD at a future low level.

المنارة للاستشارات

www.r

للاستشارات المنارة

www.m

13. Software Developer Productivity Loss Due to Technical Debt

This chapter aims to explore the consequences of TD in terms of reported wastage of software development time.

Software companies need to deliver customer value continuously, both from a short- and long-term perspective. However, software development can be impeded by technical debt (TD). Although significant theoretical work has been undertaken to describe the negative effects of TD, little empirical evidence exists on how much wasted time and additional activities TD causes. This study investigates on which activities this wasted time is spent and whether different TD types impact the wasted time differently. This study reports the results of a longitudinal study surveying 43 developers and including16 interviews followed by validation by an additional study using a different and independent dataset and focused on replicating the findings addressing the findings. The analysis of the reported wasted time revealed that developers waste, on average, 23% of their time due to TD and that developers are frequently forced to introduce new TD. The most common activity on which additional time is spent is performing additional testing. The study provides evidence that TD hinders developers by causing an excessive waste of working time, where the wasted time negatively affects productivity.

This chapter has been published as:
Software Developer Productivity Loss Due to Technical Debt - A replication and extension study examining developers' development work
T. Besker, A. Martini, and J. Bosch
Journal of Systems and Software, vol. 156, pp. 41-61, 2019.

13.1. Introduction

To survive in today's fast-growing and ever-changing business environments, large-scale software companies need to deliver customer value continuously, from both a short- and long-term perspective.

During the software development lifecycle, companies need to consider the costs of the software development process in terms of the required time and resources. In general, software companies strive to increase the number of implemented features, the overall software quality, and the overall efficiency and, at the same time, decrease the costs in each lifecycle phase by reducing time and resources deployed by the development teams.

However, software development productivity can be hindered by what is described as technical debt (TD). TD is recognized as a critical issue in today's software development industry [7] and, left unchecked in the software, TD can lead to large cost overruns, causing high maintenance costs due to internal software quality issues [95],[18],[109],[22],[24],[103],[153] and an inability to add new features [50] and even lead to a crisis point when a huge, costly refactoring or a replacement of the whole software needs to be undertaken [5].

المنـــارة للاستشارات

www.r

The TD metaphor was first coined at OOPSLA '92 by Ward Cunningham [91] to describe the need to recognize the potential long-term negative effects of immature code that occur during the software development lifecycle. Cunningham used the financial terms debt and interest when describing the concept of TD:

"*Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.*"

An additional, and more recent, definition was provided by Avgeriou et al. [4] who define TD as: "*In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability*."

This debt potentially has to be repaid with interest in the long term. Interest is the negative effect in terms of the extra effort and activities that have to be paid due to the accumulated amount of TD in the software. This may include executing manual processes that could potentially be automated or expending excessive effort on modifying unnecessarily complex code or performance problems due to lower resource usage caused by an inefficient code and similar costs [7],[151].

Software suffering from TD, however, forces the developers to perform additional time-consuming activities to be able to continue the development work with the goal of delivering high-quality software. Accordingly, an extensive amount of valuable developing working time is wasted when developers are forced to execute these additional activities due to TD in their software systems, and thus this wasted time negatively affects the efficiency and undermines the productivity of software developers. This study aims at increasing the understanding of the negative effects of experiencing TD by using weekly reporting of the wasted time over time since it is essential to have informative and cost-effective indicators to evaluate aspects of the software development processes and its product quality [154].

There are different ways of measuring software development productivity [46], and productivity is typically defined as the output divided by the effort required to produce the output. Following guidelines by Fonseca [155], we specify the operational definition [156] of developer productivity by examining the developers' reported amount of wasted time using weekly web surveys over seven weeks. Since we can determine that the amount of wasted time has a negative impact on software development duration, we define productivity as t*he ability to deliver high-quality customer value in the shortest amount of time*. This line of reasoning implies that a decrease in the amount of wasted software development time would lead to an increase in software development productivity.

Examining and quantifying the negative effects of TD plays an important role for both academia and software management practitioners in understanding and raising awareness of the magnitude of the negative effects TD has on developers' productivity. This

المنارة للاستشارات
www.r

knowledge can also help to improve software development efficiency and strategies for TD management.

The results of this study show that TD has a negative effect in terms of an extensive amount of developer working time (on average 23%) wasted due to experiencing TD during the software development lifecycle. This study also demonstrates the variance of the wasted time across the sample and, furthermore, that due to the presence of TD during the development work, developers most commonly have to perform additional testing, source code analysis, and refactoring. This study also shows that, in a quarter of the occasions where developers encounter TD, they are forced to introduce additional TD due to the already existing TD.

To the best of our knowledge, this is the first study to undertake a longitudinal examination of software developers reporting their wasted time due to TD and examining which additional activities the wasted time is spent on and also what type of TD caused the wasted time, which also adds methodological novelty to the results.

The remainder of this study is structured in seven sections: Section 2 describes the research questions, and Section 3 introduces related work. Section 4 describes the research methods in detail. Section 5 presents the research results. Sections 6 and 7 discuss the findings and threats to the validity of the study, respectively. Finally, in Section 8, conclusions and recommendations for future work will be presented.

This manuscript was originally published at the 1st International Conference on Technical Debt, held jointly with ICSE [157]. The delta of this manuscript over the prior published paper is two-sided. First this manuscript is an *extension study* of a previous paper, and secondly, this study is a *replication study* of the original study.

This new manuscript includes a value-added extension to the previous conference version of the study since that version was restricted due to space limitations. This manuscript has been extended to include additional research questions and, consequently, new findings to these research questions. The related Research section has been extended to be broader and more carefully cover additional related research publications. In the Methodology section, the novel approach of using a longitudinal research method is described in greater depth. In the Results section, we have added more subjects and more rigorous analysis by including more examples and explanations, which may increase our confidence in the conclusions. In this extended version of the previous paper, several of the figures and tables have also been refined to strengthen further the readability and understandability of the results. This manuscript also includes a validation by replication of the findings focusing on the amount of wasted time and the different encountered TD types and which additional activities this time is spent on. This replication phase confirms and strengthens the results derived in the original study. Furthermore, this manuscript includes additional in-depth analysis of how the results can be applied in practice, both by practitioners and by researchers. Finally, this manuscript also includes a potential direction for future work.

المنارة للاستشارات

www.r

13.2. Research Questions

The goal of this study, phrased as inspired by the goal-question-metric approach provided by Basili [158] is: “To analyze the consequences of TD, for the purpose of understanding, with respect to the negative effect TD has on software productivity, from the point of view of software developers and their managers, in the context of software development.” Based on this goal and more specifically, this study will examine the following six research questions:

RQ1: In what ways does technical debt waste developers’ working time?

RQ1.1: How much of software developers’ overall development time is wasted due to technical debt?

RQ1.2: Is there a significant difference between the wastage of working time due to technical debt in relation to different developer characteristics?

In this regard, the characteristics refer to different variables such as years of experience as a developer, gender, level of education, programming language, company, the age of the software, and type of software.

RQ1.3: Are there different patterns among the distributions of the wasted time over the calendar period?

The objectives of research question RQ1 (RQ1.1, RQ1.2, and RQ1.3) are to understand how much of software developers’ overall development time is wasted due to TD and whether the distribution of the wasted time varies with developer characteristics and follows any specific distribution patterns.

RQ2: Upon which extra activities is the wasted time spent?

RQ3: In what ways do different technical debt types affect the amount of wasted time?

RQ4: How often are developers forced to introduce new technical debt due to already existing technical debt?

RQ5: Is there a difference in the awareness of technical debt between developers and their managers?

The objective of this research question focuses on the awareness of the negative consequences TD has on the daily software development work and if (and in what way) the developers and managers consider the insight into the wasted time valuable.

RQ6: What are the challenges in tracking the interest of technical debt?

13.3. Related Work

In this section, we discuss related work concerning software productivity, the quantification of negative effects due to TD, how TD affects the software development productivity, and, finally, the term contagious debt.

Software productivity has been a frequently discussed subject since the beginning of software engineering research [159],[44]. In software engineering, productivity is

المنارة للاستشارات
www.r

commonly defined, from an economic point of view, as the effectiveness of productive effort measured in terms of the rate of output per unit input [44],[159],[160].

Productivity is also a measure of the quality of an output relative to the input required to produce the output. Productivity is a combined measurement of efficiency and quality. There are several constraints that can influence the software development productivity in general, such as cost, schedule, and scope [161]. Furthermore, Oliveira et al. [159] state that researchers have not yet reached a consensus on how to measure productivity properly in software engineering. However, as mentioned in the Introduction, in this study we have decided to focus on the productivity in terms of the amount of wasted software development time because developers are hindered during their daily software development work by experiencing TD within their software.

Sedano, Ralph, and Péraire [162] echo this notion by stating that "*waste is any activity that produces no value for the customer or user*" and reducing this waste of time would improve the software development productivity. In their study, they identified that software suffering from TD can lower the developer's productivity both in terms of wasted time and by causing reworking and an extraneous cognitive load.

Ernst et al. [16] surveyed three large organizations with 536 respondents and seven follow-up interviews. Based on the responses from this survey, they found that architectural decisions are the most important source of TD. This study based its conclusion on a survey in which practitioners state their perception of how the respondents perceive TD. In this study, we cannot find a quantification (reported or estimated) of the interest; there is also no explanation regarding the types of activities on which the extra time is spent.

Martini and Bosch [163] have studied the interest growth and found that some Architectural TD items are contagious, causing the interest to be not only fixed but potentially compounded, which leads to the hidden growth of interest with the potentiality of growing exponentially. Even if that study included some cases, the study did not track to the introduction of additional TD using the same level of granularity as this study.

Kazman et al. [116] present a case study for identifying and quantifying architectural debts in an industrial software project, using code changes as a proxy for calculating the interest. This study focuses on identifying the architectural roots of TD, meaning that the study does not address the cost of interest but instead aims to quantify the expected payback for refactoring. Similar to the abovementioned related research, these costs are, to some extent, based on estimated values from the interviewed architects, and, based on these assumptions, the expected benefit from the refactoring is calculated.

This study is, to a certain extent, related to a previous study [144], which also addresses the amount of wasted software development time. That previous study was based on 32 interviews and a web survey of 258 software practitioners addressing software practitioners' estimations of wasted time due to TD and also the estimations of which types of TD have the most negative impact on daily software development work and on which different activities the respondents estimate they spend this wasted time. The results of that study show that software practitioners (from several different roles) *estimate* that, on average, 36% of all development time is wasted due to TD and that

المنارة للاستشارات
www.m

Architectural TD and Requirement TD have the most negative impact, and that practitioners perceive that the majority of time is wasted on understanding and/or measuring the TD.

Later on, when studying the time spent managing TD, we found, in a study by Martini et al. [164], that the development time dedicated to managing TD is estimated to be, on average, 25% of the overall development time and generally not performed systematically

The novelty of this study's approach compared to the previous studies lies in the selection of a longitudinal research methodology adopting a different sampling strategy specifically focusing on developers and their *reported* experiences over time (compared to single estimates) by collecting repetitive observations of the same variables (e.g., wasted time) on more than one single occasion [67] and over seven weeks. The uniqueness of this extended study is also that this study validates several of the findings by conducting an additional replication study using a new and independent data set.

Compared to the previous studies, this original study had a duration of 10 months, with a longitudinal data collection phase, collecting more than 470 reported data from 43 software developers and 16 supplementary follow-up interviews with both developers and their managers. Additionally, this study also reveals new insights and interpretations on how, developers are frequently forced to introduce new TD caused by existing TD and, furthermore, reports the negative effect TD has on developers' work in terms of challenges and benefits, from both developers' and managers' sides.

To date, there is limited research available that attempts to quantify empirically how TD negatively affects software development productivity. The existing literature relating to TD and productivity states that TD becomes a constant drain on software productivity [165],[15], which leads to a slowing of the development and negatively affects productivity [7],[42],[111]. To the best of our knowledge, no previous study has employed a longitudinal empirical study (based on reported data) with the aim of understanding and quantifying how productivity (in terms of wasted software development time) is affected by TD. Our study also addresses how frequently developers are forced to introduce new TD. This topic relates to a previous study by Martini and Bosch [163] in which they found that some TDs cause other parts of the system to be contaminated with the same problem, which may lead to the non-linear growth of interest, called contagious debt.

13.4. Methodology

Triangulation is important in increasing the precision of empirical research in terms of taking different perspectives toward the studied object and thus providing a broader view [73]. To increase the validity and the reliability of the result, we have used source, observer, and methodological triangulation and also replicated the study as a last phase.

Source triangulation refers to using several sampling strategies to ensure that data are gathered at different times and in different situations. The use of more than one source of data makes the conclusion more credible since it can be drawn from several sources of information [73], [89]. In this study, we used both interviews and surveys when collecting the data.

المنارة للاستشارات
www.m

Observer triangulation refers to using more than one observer to gather and interpret data [73], [89]. This type of triangulation was achieved in this study, where at least two of the involved researchers worked together with different roles during the studies, thus enabling peer debriefing and analysis of the collected data.

Methodological triangulation refers to combining different types of data collection methods and was achieved in this study by combining both qualitative and quantitative methods [73], [89].

13.4.1. Research Design

This study is based on a longitudinal study with supplementary follow-up interviews, carried out to examine the negative impact TD has on software development, from September 2016 to June 2017. This research design was divided into seven phases, as visualized in Fig. 1. The following sections describe each phase and the related research methods used in each stage.

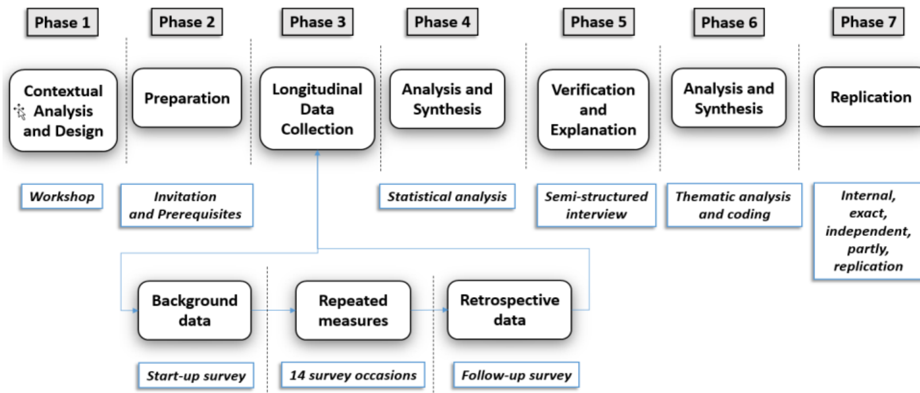


Figure 1. Visualization of the research design and research method used in each phase

13.4.1.1. Contextual Analysis and Design

First, the study was presented and discussed during a workshop with software practitioners from seven software companies within our network and with an extensive range of software development. This study’s selection of participating companies was carried out with a representative convenience sample of software professionals from our industrial partners.

This phase acted as a guide for collecting information about the studied context and selecting the most appropriate research model to use. The research team decided to base the research model on a longitudinal study together with supplementary follow-up interviews.

المنارة للاستشارات
www.r

13.4.1.2. Preparation

Secondly, an invitation to participate in the study was emailed to the participants in the workshop. Following the guidelines provided by Ployhart and Vandenberg [67], to those six companies (43 developers in total) who agreed to participate in the study, we sent out educational material intended to minimize inter-observer (all researchers communicate the same knowledge) and inter-instrument (all participants receive the same information) variability.

This educational material included different topics such as TD definitions, different terms commonly related to TD (e.g., debt, principal, and interest), TD Landscape (also illustrating what TD is not), and, finally, a short description of the different TD types. Since the definition of TD is very important and sets the context for the entire study, we specifically focused the educational material on guiding the respondents to fully understand the basic concepts of TD. In the material, we, therefore, presented three different citations to describe what is meant by TD. First, Ward Cunningham's definition was [91] offered: "Shipping first time code is like going into debt. A little debt speeds development so long it is paid back promptly with a rewrite… The danger occurs

when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on the debt," followed by Steve McConnell's definition of TD [19]:"A design or construction approach that's expedient in the short term but creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)." Furthermore, a third, shorter, definition was used: "Technical debt is a non-optimal solution in code (or other artifacts related to software development) that gives a short-term benefit but cause an extra long-term cost during the software life-cycle."

13.4.1.3. Data Collection—Longitudinal study

A longitudinal study is a research method that involves repeated observations of the same variables (e.g., time usage) on more than one occasion [67], and is conducted over time [86]. The incentive for using a longitudinal research method in this study has two principal aspects:

a) to increase the precision of reporting experienced data (in our case, not based on single estimations and single perceptions). This was achieved by studying each respondent over several weeks where the reported data could be compared. Such designs are called repeated measures designs [67], and

b) to examine the respondents' changing responses over time: Longitudinal designs have a natural appeal for the study of changes associated with development or changes over time. They have value for describing both temporal changes and their dependence on individual characteristics [67]. The longitudinal research method increases the precision of measuring and reduces inter-individual variation. This method also examines the individual's changing responses over time and provides a value for describing both temporal changes and their dependencies on individual characteristics [67].

المنارة للاستشارات

www.r

To sustain the commitment of the respondents, before starting the study, all respondents had agreed on their continuing participation with both their managers and ourselves.

The quantitative data collection during the longitudinal study was designed and hosted by an online survey service called Survey Monkey. This data collection phase included three different steps.

The first step was a start-up survey gathering descriptive statistics to summarize the backgrounds of the respondents and their companies.

Based on guidelines from [108] and to identify the population from which the subjects and objects are drawn, we studied compiled data for the participating respondents in the study. Juristo and Moreno [166] state that "the more homogeneous the elements examined in the surveys are, the better the results obtained will be," and, as illustrated in Table 1, all the respondents were relatively experienced as software developers: 56% had more than 10 years of experience, and only 7% had fewer than 2 years of experience. All respondents have a university-level education, where 82% have a master's degree. The age of the software with which the respondents worked varied, but only 2% worked with software with an age of less than 2 years, and 44% of the developers worked with software within the age range of 5-10 years. The most common system type was an embedded system, and the most common programming languages were C (40%) and C++ (21%).

The second step in the longitudinal phase used repeated measures [67]. This stage was designed to collect reported data from 43 software developers more than 14 survey occasions (i.e., twice a week for seven weeks) from October to November 2016. We collected 473 data points, and, on average, each respondent reported their data on 11 out of 14 occasions.

In this step, we emailed out an invitation to an online survey to all the respondents, twice a week (Tuesdays and Thursdays), with the goal of having equal spacing between the occasions, as suggested by Morrison [167], and, for those respondents who did not answer within one day, a reminder was emailed.

During the entire period of this phase in the longitudinal study, the participants were asked to report their answers to the three survey questions (SQ):

SQ1: How much of the overall development time have you wasted due to technical debt (TD) since last time you took the survey?

SQ2: Which extra activities were the wasted time spent on?

SQ3: What was the source of the problem for which you wasted time?

In survey question SQ1 (used for answering RQ1), the respondents reported the amount of wasted time using a value between 0-100% of their overall working time since they last took the survey. To address the potential problem with missing data from the respondents, if, for some reason, the respondents did not enter the data in one or more surveys, the respondents were always asked to report their experienced data since the last time they took the survey. This wording means that if, for some reason, the respondent did not enter the data in one or more surveys, they would enter the data from the last time the respondent took the survey. In this way, the surveys cover the full period of sampling.

المنارة للاستشارات

www.r

For survey question SQ2 (used for answering RQ2), the respondent could select between the following options on which the extra time was spent (more than one option was selectable): "Additional code analysis", "Additional testing", "Additional communication", "Additional refactoring necessary for new implementation", "Additional searching for documentation", "Implementing workarounds" and "other". If ticking the "other" option, the respondents were asked to textually describe this activity in a comment field. The listed activities were provided by Besker et al. [144].

In survey question SQ3 (used for answering RQ3), the different TD types provided by Besker et al. [144] were presented to distinguish the different sources on which the respondent had wasted time. The terms used were "Code-related issues," "Testing issues," "Architectural issues," "Documentation issues," "Requirement issues," "Infrastructure issues," and "Other." If ticking the "Other" option, the respondents were asked to textually describe this issue in a comment field.

The respondents were asked to indicate the amount of impact each of these listed issues had on their reported wasted time using a 5-point Likert Scale (Not at all - To a great extent).

The final **third step** of the longitudinal data collection phase was a follow-up survey to collect retrospective specific data from each respondent.

13.4.1.4. Analysis and Synthesis

In the fourth phase, the data collected were analyzed qualitatively, that is, by statistical analysis of the data collected from the survey answers.

For descriptive purposes, data is summarized by mean, median, and standard deviation for continuous variables, and numbers and percentages for categorical variables. The distribution of waste by the developer and by their characteristics is presented and analyzed graphically using boxplots. The strength of association between waste and various explanatory variables, such as developer characteristics and TD-types, is summarized by R-squared, describing the fraction of variance in wasted time explained by a set of explanatory variables.

Multivariable analyses of waste related to developer characteristics were also performed, aiming at finding combinations of developer characteristics associated with greater or lower waste. For this purpose, non-parametric regression trees were used. Model evaluation was performed by leave-one-out cross-validation.

The distribution of waste over time, overall, and by subject was analyzed graphically by scatterplots overlaid by smooth regression curves estimated using LOESS.

Analyses of longitudinal data were performed using logit-linear mixed effects models with the fraction of wasted time as the response variable. Activities or TD types were included as categorical explanatory variables in the models, and subject-specific intercepts and time effects were included as random effects. Random effects were included to account for subject-specific time trends and intra-individual correlations. Robust standard errors of the parameter estimates were used to obtain standard errors and confidence intervals that are robust against heteroscedasticity. Modeling was performed

المنارة للاستشارات

www.r

on a logit scale, motivated by the bounded nature of the response variable, and the effect of TD activities on waste on the original scale was computed as follows. The average effect on waste was computed by artificially changing the activity to inactive or active state for each data record, accounting for both subject effects and other concurrent activities. A confidence interval for this statistic was computed using the non-parametric bootstrap percentile method with 10000 bootstrap replicates.

Statistical analyses were performed with SPSS (version 22), SAS 9.4 (SAS Institute, Cary NC) using the GLIMMIX procedure for logit-linear mixed effects models, and R version 3.4.3 [168] using the rpart package version 4.1-13 for non-parametric regression trees [169].

13.4.1.5. Verification and Explanation

In the fifth phase, the results and conclusions acquired in previous phases were verified using supplementary qualitative semi-structured interviews. We conducted 12 interviews with developers and four interviews with their managers. All the developers had participated in the previous data collection phases, and the managers were all familiar with the study but had not actively participated in the previous data collection phases.

As suggested by Seaman [170], this study employed semi-structured interviews including a mixture of open-ended and specific questions designed to elicit not only the information foreseen but also unexpected types of information. In the interviews, the questions were planned but not necessarily asked in the same order as they were listed. This interview technique allowed for the flexibility to explore interesting insights as they emerged.

Each interview lasted between 30 and 40 minutes, and, to obtain a more accurate rendition of the interviews, all interviews were digitally recorded and transcribed verbatim. All interviewees were asked for recording permission before starting, and they all agreed to be recorded and to be anonymously quoted for this paper.

During the interviews with the developers who had participated in the quantitative data collection phase, the compiled results from their individual results were presented, and, during the interviews with their managers, an aggregated view of all the respondents from the respective company was presented. This presentation allowed the interviewees to more easily relate to the interview questions where the results of the survey were addressed. Some interview questions had a focus on corroborating certain findings that we already thought had been established during previous data collection activities, where the questions were carefully worded (avoiding leading questions) to allow the interviewee to provide fresh commentaries on them [66].

The interview questions were primarily designed to a) advance the understanding of the survey results, b) verify that the questions in the survey were understood as intended and in a uniform manner, c) confirm the results of the survey, d) help to understand the implications of the results, and e) investigate how the negative effects due to TD are communicated and managed by the companies.

المنارة للاستشارات

www.r

13.4.1.6. Analysis and synthesis

When analyzing the qualitative data collected in this thesis, a thematic analysis approach was used. Thematic analysis is a method for identifying, analyzing, and reporting patterns and themes within data, which involves searching across a dataset to find repeated patterns of meaning. The thematic analysis provides a flexible and useful research tool, which offers a detailed and complex explanation of the collected data [84].

When analyzing the qualitative data, the guidelines provided by Braun and Clarke [84] were used to conduct the analysis in a thorough and rigorous manner. The thematic analysis was conducted using a six-phase guide.

First, the audio-recorded qualitative data collected from interviews was transcribed into written form, where we were also able to familiarize ourselves with the data. The second step involved the production of initial codes from the data, where we organized the data into meaningful groups. In this phase of the analysis, a qualitative data analysis (QDA) software package called Atlas.ti was used. The third phase focused on searching for themes by sorting the different codes into potential themes and collating all the relevant coded data extracts within each identified theme. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the citation "Maybe you have to encourage the developers a bit, to get the data" was coded as "Willingness to input data" in the theme "Measuring Wasted Time Aspects." To ensure that the coding was performed in a consistent and reliable fashion, triangulate the interpretation of the data, and avoid bias as much as possible, two authors synchronized the output of the coding, following guidelines provided by Campbell et al. [85].

The fourth phase focused on the revised set of candidate themes involving the refinement of those themes. The refinement focused on forming coherent patterns within the themes. Otherwise, we revised the themes or created a new theme. The fifth phase focused on identifying the essence of each theme and determining what aspect of the data each theme captured. This phase also stressed the importance of not just paraphrasing the content of the data extracts, but also identifying what is interesting about them and why. The final phase of the thematic analysis took place when we had a set of fully developed themes and involved the final analysis.

المنارة للاستشارات

www.r

Based on the research taxonomy, a coding scheme containing four broad themes and 22 individual codes was developed. Fig. 2 shows the outcome of the analysis process, where the mapping between different hierarchical categories and individual codes are graphically presented.

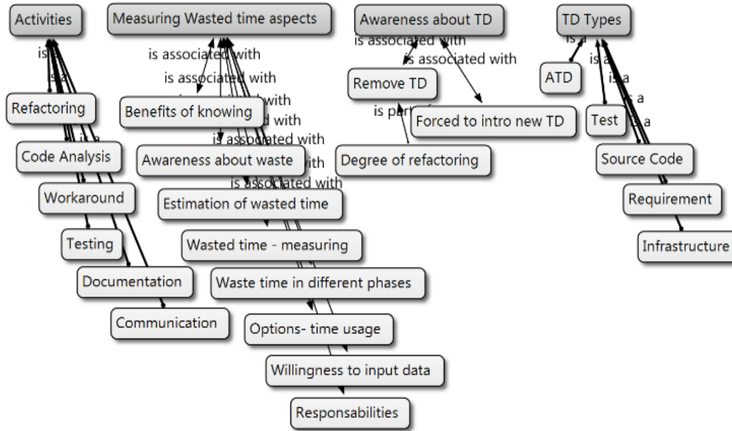


Figure 2. Coding Scheme

TABLE 1 - CHARACTERISTICS OF RESPONDENTS

Individual-level	No. of respondents	(%)	Company level	No. of respondents	(%)
Experience			Software system type*		
< 2 years	3	(6.98%)	Embedded system	29	(67.44%)
2-5 years	7	(16.28%)	Real-time system	14	(32.56%)
5-10 years	9	(20.93%)	Data management system	5	(11.63%)
> 10 years	24	(55.81%)	System Integration	2	(4.65%)
			Modeling and/or simul.	2	(4.65%)
Educational level			Data analysis system	7	(16.28%)
Master's	35	(81.40%)	Web 2.0 / SaaS system	1	(2.33%)
Bachelor's	7	(16.28%)	Other	1	(2.33%)
No. Univ. education	0	(0.00%)	System Age		
Other:	0	(0.00%)	< 2 years	1	(2.33%)
Ph.D.	1	(2.33%)	2-5 years	10	(23.26%)
			5-10 years	19	(44.19%)
Gender			10-20 years	11	(25.58%)
Male	36	(83.72%)	>20 years	2	(4.65%)
Female	7	(16.28%)	Programming Language		
			C	17	(39.5%)
			C++	9	(20.9%)
			Java	4	(9.3%)
			Python	7	(16.28%)
			Ada	3	(6.98%)
			Other	3	(6.98%)

* More than one option was selectable.

13.4.1.7. Internal Exact Independent Partly Replication

Replication plays a key role in empirical software engineering [68], and is proposed as an important means of increasing confidence and assessing reliability in the result [69], [70].

As illustrated in Fig. 1, the original study consists of six different research phases, followed by a seventh phase in the replicated study. After the first sixth research phases, we were able to answer all stated research questions. The replicated study aimed at

المنارة للاستشارات

www.r

answering three of the identical research questions used in the original study. After the results from the replicated study were derived, these results were compared with the results from the original study to either confirm or disconfirm the original findings.

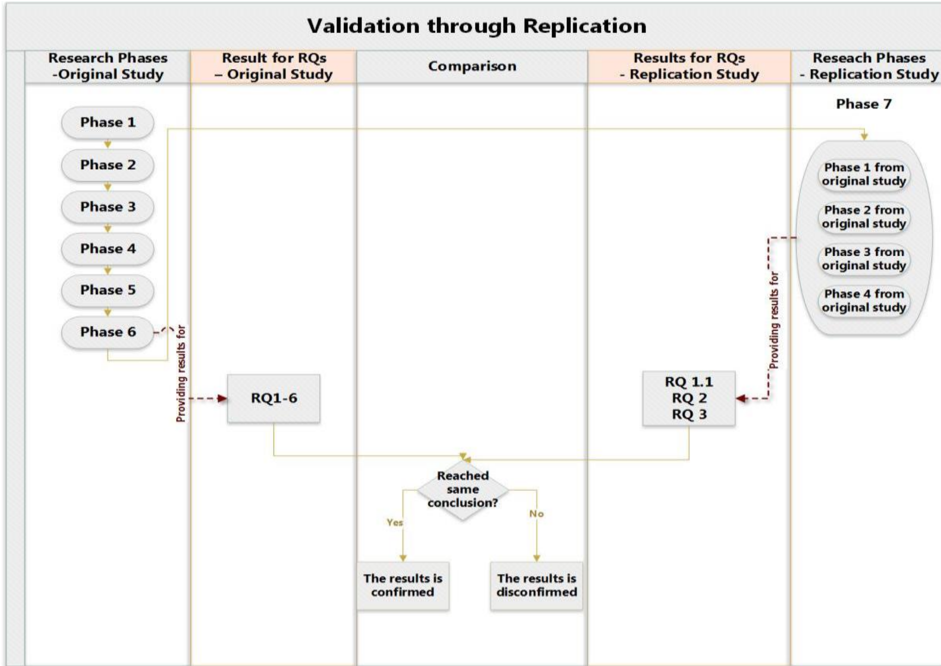


Figure 3. Integration of Replication study

As described above, in the seventh phase in this study we have conducted an internal exact independent partly replication of the study, where the *Internal* reflects that the replication team is the same as in the previous phases. The categorization *Exact* reflects that the procedures are followed as closely as possible to determine whether the same results can be obtained. To gain more insight into the original results, this categorization also includes replications that are modified to some extent by, for example, altering the subject pool or other conditions. The categorization of *Independence* reflects that, during the replication phase of the study, we deliberately varied some aspects of the conditions when collecting the data [68]. The categorization *Partly* points out that not all of the research areas and research questions are replicated. More specifically, this phase aims at replicating the findings for RQ1.1, RQ2, and RQ3 addressing the overall amount of wasted time due to TD and also the extent of encountering different types of TD.
Due to restrictions on how this replication study could be carried out by the involved company, and what data was feasible to collect, the datasets in this replication study do not include enough information to allow for replications of all original research questions. For instance, the replication study did not include any follow-up interviews with the participating developers and their managers. This limitation resulted in RQ5 and RQ6 not being replicated in this phase of the study, and they were, therefore, omitted.

المنارة للاستشارات
www.r

The design of this phase follows the guidelines proposed by Carver [171] for reporting replication studies. The motivation for conducting this replication phase was to validate the results from the original study by changing the participant pool and the way we collected data to gain additional confidence that the original results were not the result of, for example, a data collection bias or the selection of the study design. Below, we describe each activity for the replication study, using the same phases as we used in the original study, to illustrate similarities and/or differences between the two sets of studies.

In the first phase (in comparison with section 4.1.1) of the replication study, this part of the study was presented to one specific software company in Germany, with an extensive range of software development. The company name has been anonymized for confidentiality reasons. The research team decided with the contact persons at the company on a suitable research model to use in this replication phase of the study.

During a second phase (in comparison with section 4.1.2) in the replication phase, the contact persons in the company invited developers to voluntary participation in the study. Similar to the original study, all the participants were provided educational material about the research topic.

The developers were quite experienced, where 21% had worked for more than 10 years in software development, and only 21% had fewer than five years of experience. Fully 56% of the respondents had a bachelor's degree, and only two developers had no formal higher education.

We cannot share the rest of the characteristics of the developers and the company due to confidentially reasons (such as gender, software system type, and programming language).

The third phase (in comparison with section 4.1.3) consisted of two steps of data collection using two sets of surveys. The first survey gathered similar descriptive statistics to summarize the backgrounds of the participants as were collected in the original study. The second step collected data using a similar longitudinal research design (one survey per week). This stage was designed to collect reported data from 47 software developers. However, the replication survey was designed slightly differently, compared to the survey that was used in the original study. Each week, the respondents were asked to report on *one specific major work item* within their ongoing project and both how much time the respondents spent on this item and how much time was wasted due to experiencing TD for this specific item. More specifically, the participants were asked to report the share of their total development time spent on the specific work item and the share of that time they wasted due to TD as well as what extra activities the wasted time was spent on and the source of the problem for which they wasted time on this work item?

The fourth phase (in comparison with section 4.1.4), the analysis of the collected data in the replicated study has been analyzed in quantitatively, that is, by interpreting the numbers collected from the survey answers. The replicated phase did not include steps similar to the fifth and sixth phases we conducted in the original study, where we conducted supplementary qualitative semi-structured interviews and analysis of those.

In total, we received data from 177 different working items, and the results of the replicated study and its comparison with the original study are presented in Section 6.5.7.

المنارة للاستشارات
www.m

13.5. Results and Findings

The following subsections present the results for the research questions presented in Section 2, and the results are grouped according to each research question.

13.5.1. Wasted time

The first set of questions (RQ1.1, RQ1.2, and RQ1.3) focused on how much of software developers' overall development time is wasted due to TD and whether the distribution of the wasted time varies with developer characteristics and follows any patterns.

13.5.1.1. Wasted time (RQ1.1)

During the longitudinal data collection phase, 43 developers reported their wasted working time due to experiencing TD twice a week for seven weeks (in total 473 data points). On average, each developer reported 10.7 times out of 14 possible occasions (with a median of 12 and standard deviation of 3.9 times) with the average time interval between the reporting occasions of 3.1 days, with a median of 2.7 and standard deviation of 1.3 days (excluding Saturdays and Sundays).

The single most striking observation to emerge from the data was that the respondents reported that, on average, **23.1%** of all software development time is wasted due to TD, with the standard deviation of 21.1% and a median value of 17.13%.

When calculating the average amount of wasted time, the different interval lengths between the occasions were taken into account. This meant that, for example, when a developer reported 33% waste for a three-day period following a reported waste of 40% for a 10-day period, the average wasted time was calculated as 38.38%.

Turning to the distribution of the reported wasted time, Fig. 4 shows a histogram of the respondents and their wasted time. Most respondents wasted between 0% and 10% of their working time. It is interesting that six respondents reported a wastage of more than 50% of their working time.

To put the above numbers into context, Fig. 5 shows an overview of the distribution of the wasted time as a function of calendar time. The mean wasted time remained quite stable over the entire study period at about 25%.

المنارة للاستشارات
www.r

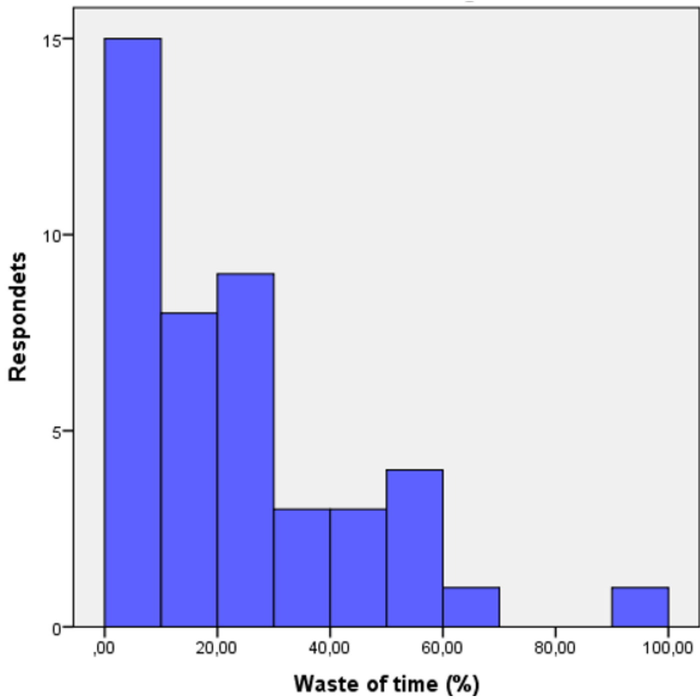


Figure 4. Distribution of the reported wasted time

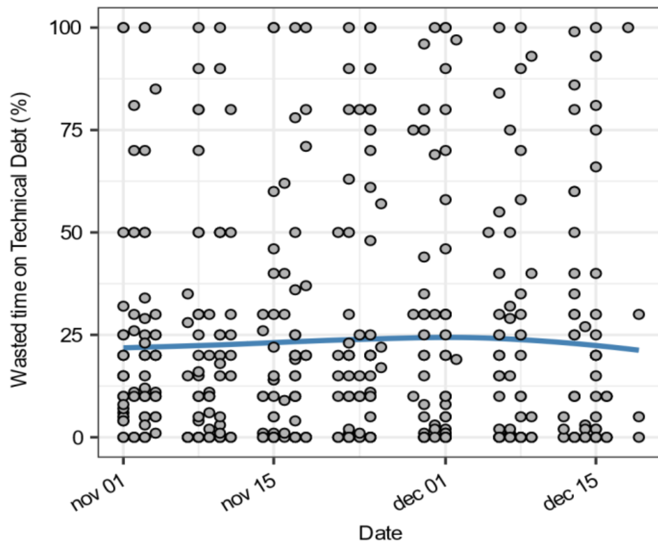


Figure 5. Wasted time on technical debt as a function of calendar time. The blue curve, presenting mean waste as a smooth function of time, was estimated using LOESS.

المنارة للاستشارات

www.r

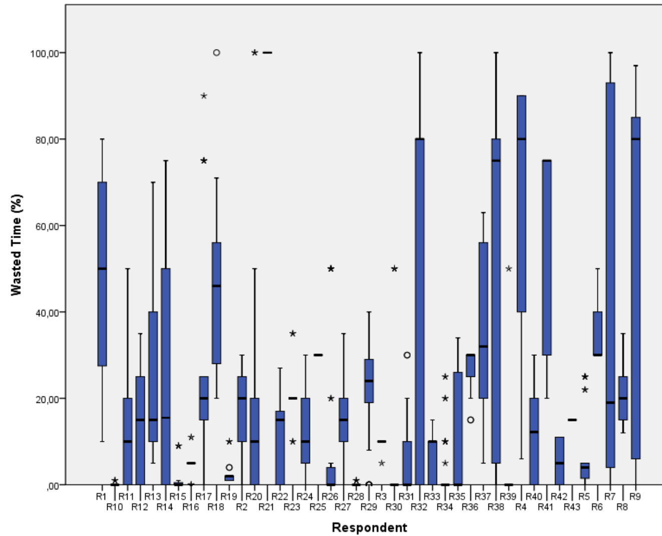


Figure 6. Distribution of the wasted time for each respondent

Fig. 6 shows the distribution of each respondent's reported wasted time illustrated by a boxplot. Looking at the comparison of medians (the bold horizontal black line in each box) among the different respondents in the figure, it is apparent that the different respondents waste different amounts of time due to experiencing TD. The figure also demonstrates that there are different distances between the median values and the upper and lower quartile among the different respondents, which indicates that, among the respondents, there are variances in terms of how consistently the wasted time is reported. Some respondents' amounts of reported wasted time vary greatly over time, while other respondents' amounts of reported wasted time are more consistent and concentrated.

13.5.1.2. Characteristics (RQ1.2)

When examining the distribution of the wasted time with respect to different characteristics, we focused on the different variables: (a) years of experience as a developer, (b) gender, (c) level of education, (d) programming language, (e) company, (f) age of the software, and, (g) type of software.

The used variables were the same variables that were collected and assessed in the first step in the data collection phase (e.g., the start-up survey).

The percentage of wasted time versus related to various subject characteristics is presented in Fig. 7 and Fig. 8. As illustrated in Fig. 7, three variables showed substantial differences with respect to the reported amount of wasted time:

- The company, explaining 20.4% of the variance in wasted time, with an average waste ranging from 18.1% in company A to 51.5% in company G (Fig. 7a).
- Software age, explaining 17.4% of the variance in wasted time, with an average waste ranging from 15.3% (software 5 - 10 years old) to 55.3% (software > 20

المنارة للاستشارات

www.r

years old, n = 2) (Fig. 7b).

- Type of software, explaining 33.3% of the variance in waste time. The wasted time was below average for Modelling and Simulation Systems, Real-Time Systems and Embedded Systems, and more than twice the average for Data Management Systems, System Integration Web 2.0 / SaaS Systems (Fig. 7c).

As illustrated in Fig. 8, there were only small variations in the reported amount of waste time with respect to experience, gender, level of education and programming language.

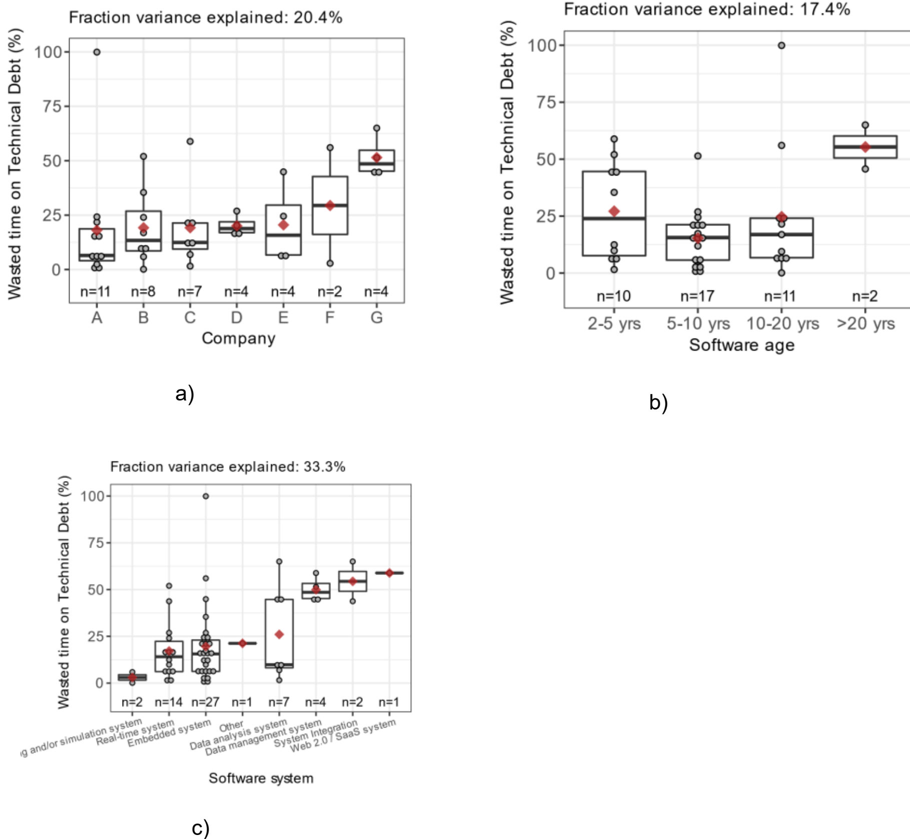


Figure. 7 (a, b, c): Percentage wasted due to technical debt vs. (a) company, (b) software age, and (c) software system. Circles represent individual data points, binned into 50 distinct intervals along the y-axis. The red diamonds show the mean within each group. For a software system, individuals may appear in multiple groups. Means and fraction variance explained are computed by ordinary least squares regression, taking concurrent software systems into account.

المنارة للاستشارات
www.r

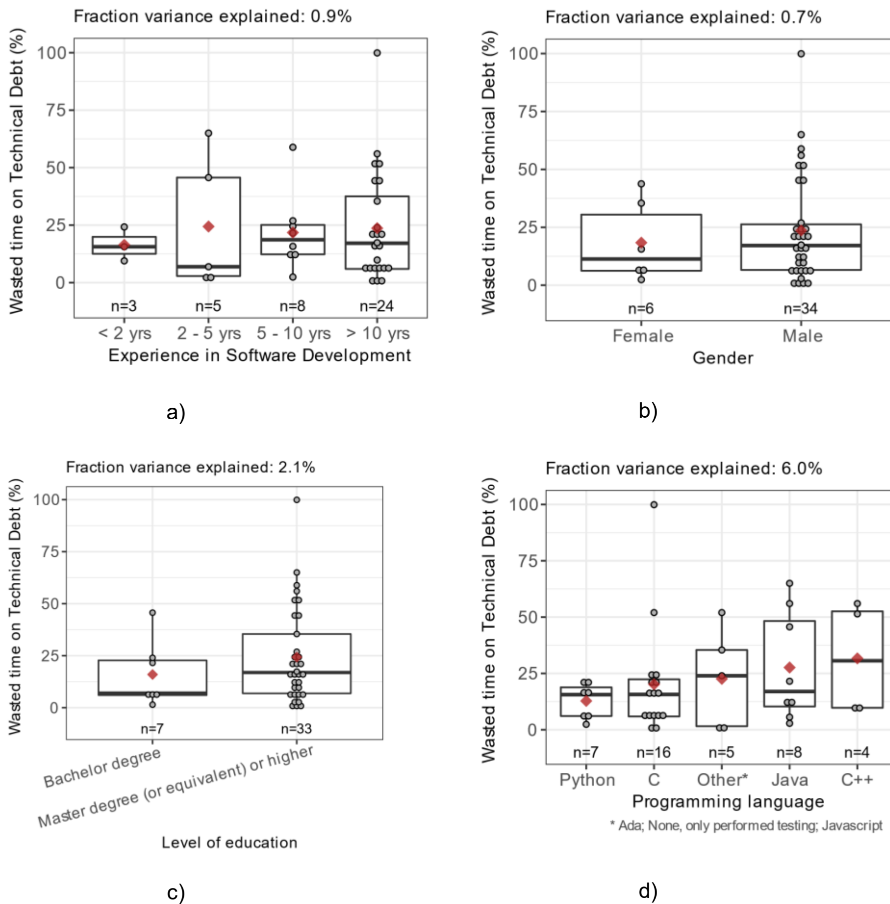


Figure. 8 (a, b, c, d): Percent wasted time on technical debt vs. experience in software development (a), gender (b), level of education (c), and programming language (d). Circles represent individual data points, binned into 50 distinct intervals along the y-axis. The red diamonds show the mean within each group. For the programming language, individuals may appear in multiple groups. Means and fraction variance explained are computed by ordinary least squares regression, taking concurrent programming languages into account.

Developer characteristics were further evaluated in multivariable analyses using non-parametric regression trees, aiming to find combinations of characteristics associated with greater or lower waste. However, only trivial models consisting of a single variable were found, as adding more variables resulted in increased cross-validation error. This is probably due to the limited sample size and more specifically to the small number of individuals in subgroups with high waste.

المنارة للاستشارات

www.r

13.5.1.3. Distribution (RQ1.3)

When examining each respondent's distribution of the wasted time over the study period, we noticed that the variations in mean level and trend during the study period differed between the respondents. All respondents showed an individual distribution of the wasted time during the study period, but, when examining all different distributions, we could identify four main distribution profiles of the reported wasted time which were generally common to all of the respondents' reported data. The four identified profiles are associated with the pattern of the distribution of the wasted time and labeled: *Fluctuating*, *Periodical*, *High*, and *Low*. Examples of these profiles are illustrated in Fig. 9. For some developers, it was evident that the wasted time varied *periodically* over time, meaning that, within a sub-period, the distribution followed a *low*, *high*, or *fluctuating* pattern, but, at some point, this pattern changed. The fluctuating profile demonstrates that, over time, the wasted time did not show any clearly discernable time-related pattern and included both high and low amounts of wasted time. The Low and High profiles illustrate a wasted time that is largely consistent over time, even if some peaks can be recognized.

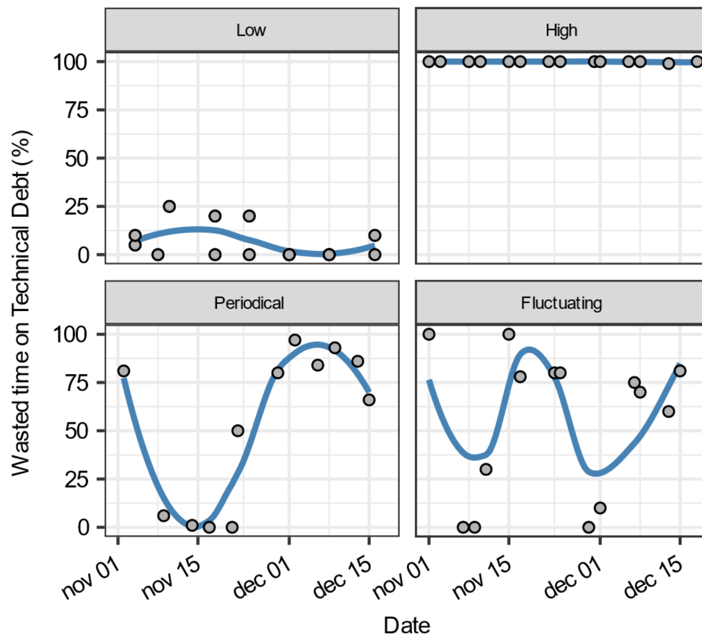


Figure 9: Different distribution profiles of the wasted time. The blue curves, presenting mean waste as a smooth function of time, were estimated using LOESS.

The most common pattern among the respondents was the Low profile, and the less common pattern was the High profile. The distribution between the Fluctuating and the Periodical patterns was largely equally divided.

المنارة للاستشارات

www.r

Finding 1: *Almost a quarter of all developers' working time is reported as wasted due to having TD.*

Finding 2: *Company and System types have the strongest impact on the amount of wasted time.*

Finding 3: *Even if the distribution of the wasted time varies over time for individual developers (following different identified patterns), the overall distribution of the wasted time for all developers and over time is largely consistent.*

13.5.2. Additional activities

The next research question (RQ2) explores the different activities on which the wasted time is spent and also whether the amount of the wasted time relates to any specific activity.

When developers encounter TD during their software development work, they are forced to perform supplementary actions. Accordingly, these different activities would not have been necessary if the TD were not present.

During the longitudinal data collection phase, the respondents were asked to report the additional activities on which the wasted time was spent during each occurrence. For each of the 473 reporting occurrences, the respondents selected the activities on which the wasted time was spent from a list of pre-defined options (listed in Section 4.1.2).

The distribution of wasted time across different activities is presented in Fig. 10, with activities sorted according to mean waste per activity. As illustrated in this Figure, the mean wasted time was greatest when performing *Additional Testing*, with a mean wasted time of 43.1%, followed by *Additional Refactoring* (mean waste 42.8%) and *Additional Code Analysis* (mean waste 37.9%). The activity with the weakest association to the wasted time is *Additional Communication*, with a mean waste of 28.8%. The "*None*" option was mainly chosen when no time at all was wasted (mean waste 5.4%).

The marginal effect of each activity, accounting for concurrent activities and subject effects through mixed effects models, is presented in Table 2. The activity with the strongest effect on wastage of time was again performing Additional Testing, associated with a waste increase of 12.4 percentage units (p.u.) (95% CI 8.7 to 17.3 p.u.), followed by Additional Code Analysis (mean waste increase 11.7 p.u) and Additional Refactoring (mean waste increase 10.3 p.u.). Again, performing Additional Communication was associated with little additional waste (mean waste increase 4.1 p.u., 95% CI 0.9 to 8.0 p.u.). This means that having to perform Additional Code Analysis increases the average amount of wasted time by 12.4%, compared to if no additional code analysis had to be performed. When selecting among the different activities the wasted time was spent upon, the respondents also could enter an additional activity manually in a text field that was

المنارة للاستشارات

www.r

not predefined, and, interestingly, no other additional activities caused by the present TD were added here by the respondents. This implies that the six listed activities cover most of the extra activities on which the time is wasted due to experiencing TD.

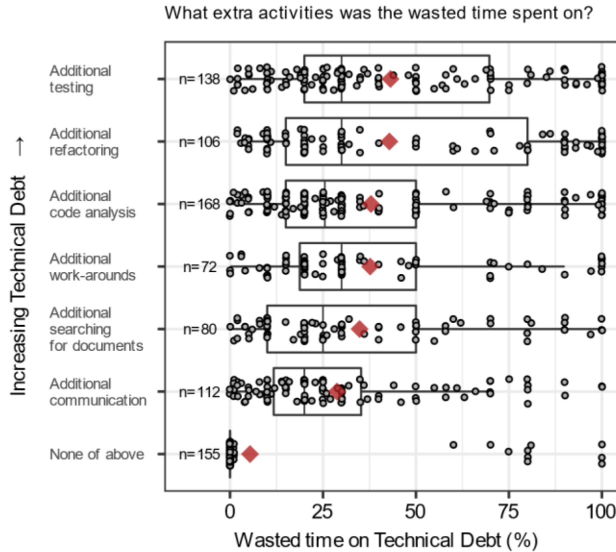


Figure 10: Wasted time due to technical debt vs. Activities. Circles represent individual data points, binned into 50 distinct intervals along the y-axis. The red diamonds represent the mean waste for each activity.

Table 2
The average effect of TD activities on wasted time.

What extra activities was the wasted time spent on?	Estimated effect on waste increase* (95% CI)**
Additional testing	12.4 (8.7; 17.3)
Additional code analysis	11.7 (7.8; 16.0)
Additional refactoring	10.3 (6.4; 15.2)
Additional searching for documents	6.5 (2.3; 11.6)
Additional work-arounds	6.3 (1.7; 10.1)
Additional communication	4.1 (0.9; 8.0)

* Estimated effects are presented in percentage units.
** 95% confidence intervals were computed using the non-parametric bootstrap percentile method, using 10,000 bootstrap replicates.

Finding 4: *The activity "Additional testing" has the strongest association with the wasted time followed by conducting "Additional source code analysis" and "Additional refactoring."*

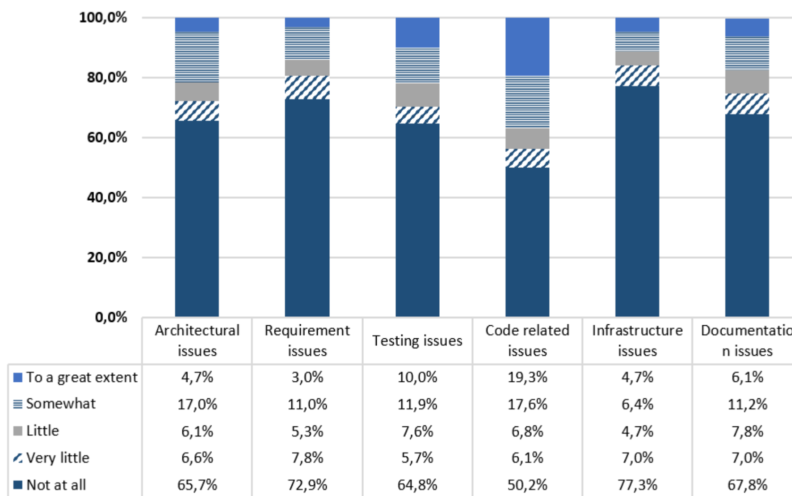
المنارة للاستشارات
www.m

13.5.3. Technical debt types

Since there are several types of TD [14], and these different TD types could have different levels of negative impact on the amount of the wasted time, and these different TD types could potentially have different levels of negative impact on the amount of the wasted time, in this third research question (RQ3), we sought to explore the ways in which these TD types impact wasted time and also which TD type has the most negative impact on the wasted time from a developer's perspective. For each reporting occasion, during the longitudinal data collection phase, the respondents ranked the level of negative impact different listed TD types had on the reported wasted time, using a list of different TD types. For each listed TD type, a 5-point Likert ranking scale was set from "Not at all" to "To a great extent."

From the data in Table 3, it is apparent that a significant proportion of the TD encountered is related to source code, where 19.3% of the code-related TD is encountered "To a great extent."

TABLE 3 - The Likert scale of each encountered TD type



	Architectural issues	Requirement issues	Testing issues	Code related issues	Infrastructure issues	Documentation issues
To a great extent	4,7%	3,0%	10,0%	19,3%	4,7%	6,1%
Somewhat	17,0%	11,0%	11,9%	17,6%	6,4%	11,2%
Little	6,1%	5,3%	7,6%	6,8%	4,7%	7,8%
Very little	6,6%	7,8%	5,7%	6,1%	7,0%	7,0%
Not at all	65,7%	72,9%	64,8%	50,2%	77,3%	67,8%

Fig. 11 illustrates each studied TD type and its relation to the reported amount of wasted time. A positive trend was observed between increased levels of encountering the different TD-types and increased average waste—meaning that the more of each TD type the developers encounter, the more time they waste—although a monotonic trend was not observed for all TD types. The strongest association with wasted time was observed for *Testing issues*, explaining 21.2% of the variance in wasted time, followed by *Code-related issues* and *Architectural issues*. Only a weak association was observed between wasted time and *Requirement issues*, explaining only 9.0% of the variation in

المنارة للاستشارات
www.r

waste. We, therefore, suggest that the association of Requirement TD and the amount of wasted time be investigated further in future studies.

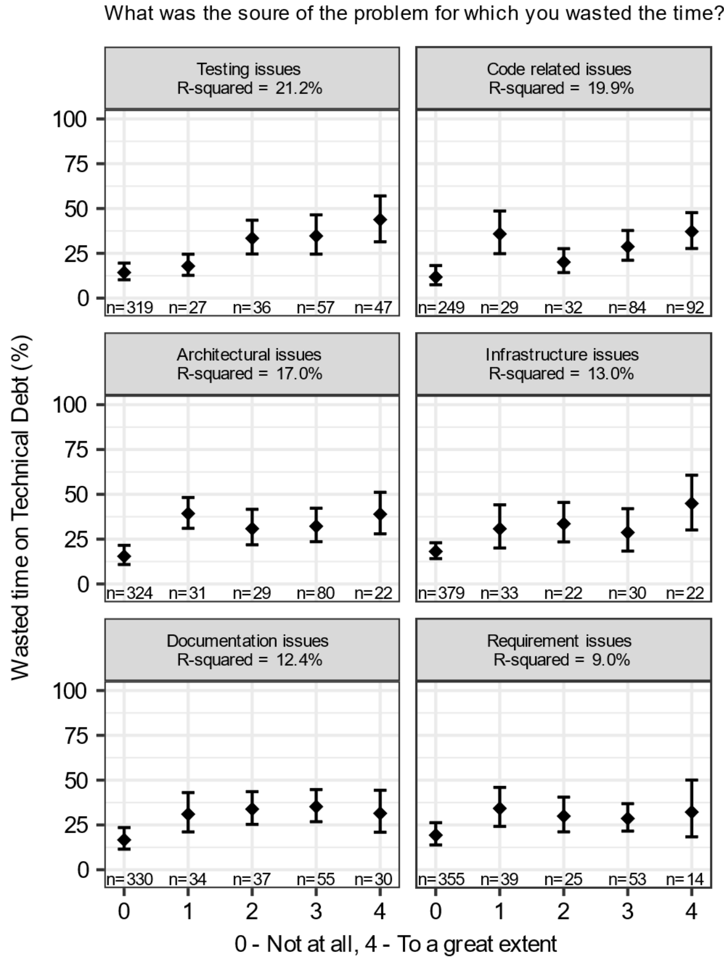


Figure 11: Wasted time on technical debt vs. technical debt type. The black diamonds represent the mean for each level on the Likert scale, error bars present 95% confidence intervals for the mean.

Finding 5: *The TD type "Testing issues" has the strongest association with the wasted time, followed by "Code-related issues" and "Architectural issues," with increased level of negative impact associated with increased amount of wasted time. Only a weak association between Requirement TD and amount of wasted time was observed.*

المنارة للاستشارات
www.r

13.5.4. Introducing new Technical debt

Sometimes, developers are forced to introduce new additional TD due to already existing TD. The interest payment could take place in the form of, for example, introducing new shortcuts and maintenance obligations taken as the developer tries to fix the prior debt.

This research question (RQ4) aims to address the amount of additional TD developers are forced to introduce due to present TD.

The result in Fig. 12 graphically illustrates that in 24% of all the reported occasions the developers reported that they were, to some extent, forced to introduce additional TD. Within these 24% reported occasions, on 13% of the occasions the respondents reported that they were forced to introduce additional TD "To a very little extent," and 3.6% reported "To a little extent," and 4% "To some extent," and 3% reported that they were forced to introduce additional TD "To a great extent."

When performing a detailed analysis of each reported occasion where the respondents were forced to introduce new additional TD due to already existing TD "To a great extent," the result shows that the encountered TD types for these occasions were Test TD (in 53% of the occasions) and Source Code TD (in 47% of the occasions).

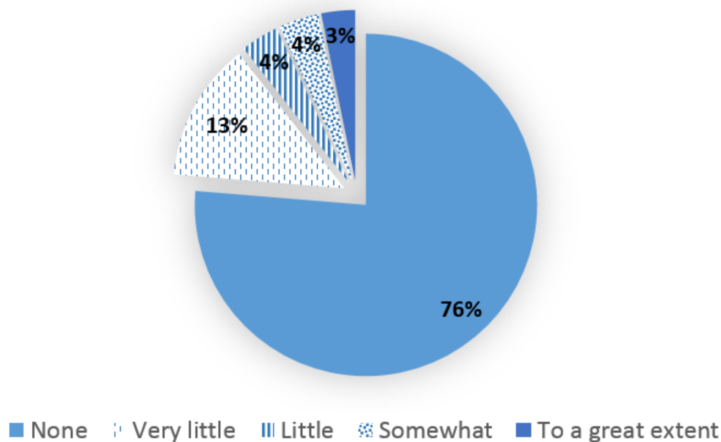


Figure 12. Introduction of new TD

The majority of the interviewees explained the reasons why they were forced to introduce additional TD in terms of "time pressure."

This expressed time pressure was commonly described both in relation to the implementation of the solution, but also that it caused other activities to suffer, such as performing sufficient testing- or updating related documentations.

For example, one interviewee claimed,"*Usually, it takes a longer time to make the correct solution. It is more or less always a time question. Often, when you introduce technical debt, it's because something had turned up. Which was not quite the way that we thought it was when we planned how to do the software*." Moreover, in discussing this issue,

المنارة للاستشارات

www.r

another interviewee said, "*You implement suboptimal solutions because you are in a hurry. Plus, we don't find the time to use the test tools we have and write tests for everything.*"

Finding 6: *In a quarter of all occasions of encountering TD, developers are forced to introduce additional TD due to already existing TD, potentially causing contagious debt.*

Finding 7: *Encountering Test TD and Source Code TD forces the developers to introduce additional TD to the largest extent.*

13.5.5. Awareness and Benefits

This research question (RQ5) focuses on the awareness of the negative consequences TD has on the daily software development work and whether (and in what way) the developers and managers consider the insight of the wasted time valuable.

Apart from when participating in this study, none of the developers explicitly measured, tracked, or reported their wasted time but still considered themselves to have a high level of awareness regarding the amount of time they wasted due to TD.

Initially, during the interviews with each of the developers, we asked them how much time they estimate they waste in general, and after that, we showed them their results from their average reported wasted time from the longitudinal study. When we presented the individually reported wasted time for each developer, all developers acknowledged their reported amount of time. As one interviewee stated, "Yes, so we thought there would be some time, so it's not that we're shocked by it [22% wasted time], but it could have been worse," while another developer commented, "To me, it's a natural part that you waste 25%, and I think it's quite reasonable to spend so much time maintaining old code."

On the other hand, during the interviews with the developers' managers, the general level of awareness of the amount of time developers waste due to TD was considerably lower. As one manager observed, "As a manager, if I had data telling me that people are wasting around 25% of the time they have available for developing, I would for sure like to know that because that's unacceptable... If I knew, I would be able to do something about it, or at least to raise the problem."

These quotes also highlight the different ways developers and managers seem to appraise the amount of development time that is reasonable to waste due to TD.

Overall, both developers and managers considered the benefits of knowing the amount of wasted time similarly. The benefits were described by the developers as the quantified wasted time potentially helping to detect and to predict the need for additional quality improvements.

Some developers also highlighted the benefits of being able to improve forecasting and capacity planning and being able to justify change and thereby motivate their managers.

المنارة للاستشارات

www.r

For example, one developer stated: "Yes, it would be useful when you do the estimation of when you start a project and when you finish it.... So I could use it to predict my baselines in my delivering."

One developer in the study also described the benefits of tracking the wasted time as a useful tool when implementing a new type of management strategy with the goal of decreasing the negative impact of TD. "You could combine it [the reporting of the wasted time] with working in another way. Because you can use it for tracking…We should also change the way that we actually make new things to try to work without creating new technical debt. And then you use the tracking so if these working methods actually work." Furthermore, when discussing the quality issues, an interviewee claimed, "If I had a huge amount of wasted time, it also shows that we have a lack of quality… I think it would be a tool that could help us improve our quality."

Moreover, the managers emphasize the benefits of quantifying the amount of the wasted time in terms of being able to discuss and identify various causes that adversely affect the development work. For example, one manager claimed, "But, as a manager, I thought it was interesting because I want there to be as few barriers to my team as possible. And this is a form of obstacle. And sometimes when you ask people 'what's the obstacle to you?' then it almost becomes more an emotional question than it becomes a fact."

Finding 8: *Developers have a higher awareness than their managers of how much time is wasted due to TD, and the developers and their managers seem to appraise the amount of development time that is reasonable to waste due to TD in different ways.*

Finding 9: *Both developers and managers described the benefits of knowing the amount of wasted time in a similar manner. The amount of wasted time was found to be a useful indicator of the software quality, and it could also be used for improving forecasting and better capacity planning and assisting the communication between*

13.5.6. Challenges of tracking TD interest

This question (RQ6) highlights the interaction between developers and managers with regard to the challenges of tracking the interest of TD.

Overall, even though developers consider themselves to benefit from making the amount of wasted time evident, most of the developers did not have a positive attitude toward continuously reporting the wasted time to their managers. Even if the interviewed developers generally claimed that the specific reporting task took them only a couple of minutes for each reporting occasion, several interviewees argued that this reporting task would require unwanted additional time and effort.

Similarly, even if the managers consider the benefits of knowing the amount of wasted time to be important, the managers did not explicitly request this information from the

المنارة للاستشارات

www.m

developers, and, in general, they did not have a positive attitude to asking the developers to report their wasted time.

Several managers expressed their unwillingness to introduce new reporting tasks to the developers, and one manager described the fear of causing extra stress for the developers. “There is a certain fear of reporting. It will almost become a negative spiral of it eventually. We have little stress-related sick leave. If you get that, because of such a system, you probably have not achieved that much.” Another manager echoed this notion, describing the attitude toward introducing reporting of wasted time to the developers. “Even though I see the value of this kind of data, it's nothing I'm going to force anyone to report, but if people are interested in continuing this here, I'm very welcome from my side, but it may be on an interest-based basis.”

Even if all interviewees were familiar with the concept of TD and its related negative effects, this knowledge was not put into practice, and the lack of having an overall strategy for managing TD was evident.

> **Finding 10:** *The willingness to quantify the wasted time is a major challenge, since developers and managers do not, in general, have a positive attitude toward implementing additional reporting.*
>
> **Finding 11**: *None of the interviewed companies had a clear strategy on how to track and address the wasted time.*

13.5.7. Results of the Replication study

The motivation to carry out this replication part of the study is to investigate further whether the previous results demonstrate that the findings can be repeatedly generated and thus the original findings were not an exceptional case. In particular, the aim is to broaden the results obtained in RQ1.1, RQ2, and RQ3 by investigating the same research areas but with other independent data sets, as described in section 4.1.7.

Each sub-section will first report the results from the replicated study and then present a comparison of its results with the results of the original study.

13.5.7.1. Replication of result addressing wasted time due to technical debt

Research question RQ1.1 addresses how much of software developers’ overall development time is wasted due to technical debt. The replicated study focuses on how much time was wasted on specific working items.

In total, we have analyzed data from 177 reported working items from 47 developers at the same company but working on different projects and with different products.

المنارة للاستشارات
www.r

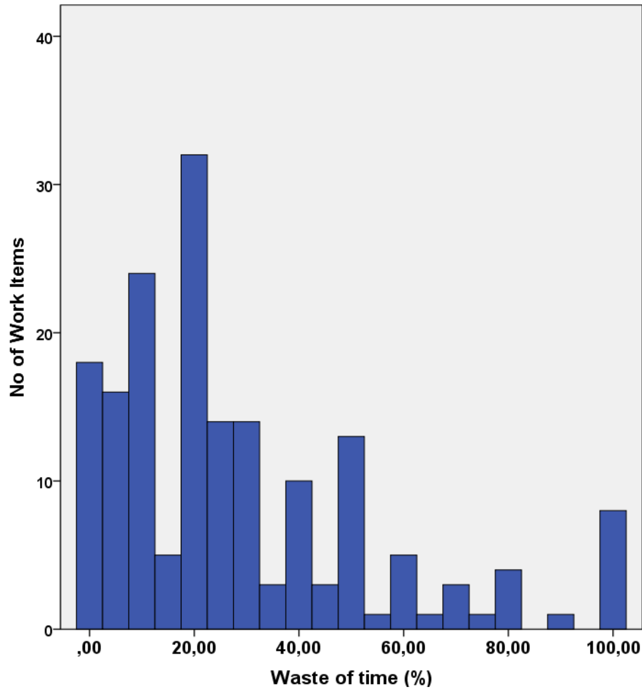


Figure 13. Distribution of the reported wasted time from the replicated phase

The replicated study found that, on average, 28.51% of the software development time for the reported working tasks is wasted due to TD, with the standard deviation of 25.37% and a median value of 20.0%. Fig. 13 shows a histogram of the reported wasted time for each work item. Most work items wasted, on average, 20% of their time due to experiencing technical debt.

Even if both data sets from the original and the replicated study are large enough to support comparison, they do not permit comparison using the same statistical methods since they are based on different variables. However, the result of the two studies can be examined together by studying the result of the reported amount of wasted time.

In the replicated study, the respondents were asked to report on the wasted time for the work item they spent most of their working time on since the last time they took the survey. However, since the respondents potentially could perform other working tasks during the period (for which we do not know the amount of wasted time), we cannot generalize their reported waste of time for the full period. Our analysis of the time the respondents spent on the reported work items shows that the respondents spend on average 57.63% of their working time on each of reported work item.

Despite the different used variables, one might intuitively expect that the amount of wasted time due to TD should be quite similar in both of the studies. In the original study, the respondents reported that, on average, 23.1% of all software development time is

المنارة للاستشارات

www.r

wasted due to TD. This result is a slighter lower share of wasted time than the reported average of the replicated study, where the respondents reported that, on average, they waste 28.5% on each of the reported tasks. However, by combining results from the original study with results from the replication study, we conclude that Finding 1 stating that "*Almost a quarter of all developers' working time is reported as wasted due to having TD*" is valid and strengthened.

13.5.7.2. Replication of result addressing different activities

This part of the replication study aims at replicating the findings from RQ2 by exploring the different activities on which the wasted time is spent and also whether the amount of the wasted time relates to any specific activity.

The distribution of wasted time across different activities from the replication study is presented in Fig. 14 where the activities are sorted according to mean waste per activity. By studying the ranking of activities with respect to average waste of time in this figure, it is evident that this data differs noticeably from the data in the original study. For instance, in the replicated study, the activity of performing "*Additional code analysis*" has the strongest association to the wasted time whereas, in the original study, performing "*Additional Testing*" has the strongest association. In the replication study, the additional activity of performing "*Additional Testing*" is ranked quite differently since it has the weakest association with the wasted time (except for the "other activity option").

However, one should note that the mean value of each of the activities is fairly close to the original study, except for the mean value of "*Additional Testing*."

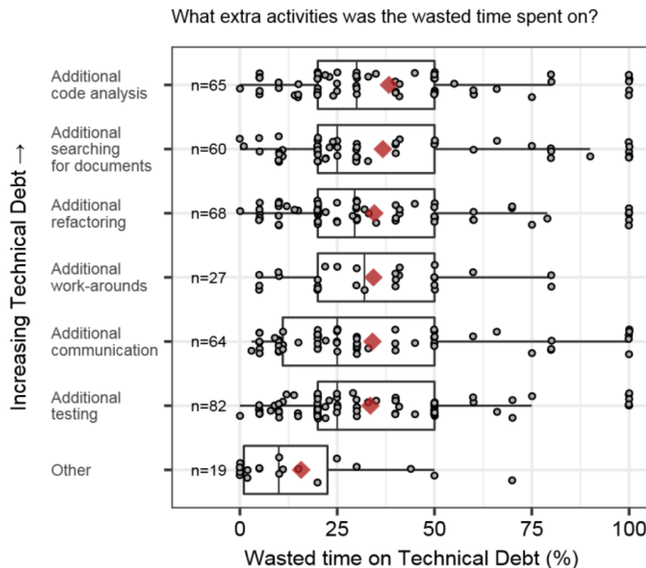


Figure 14: Wasted time due to technical debt vs. activities for the replicated study. Circles represent individual data points, binned into 50 distinct intervals along the y-axis. The red diamonds represent the mean waste for each activity.

المنارة للاستشارات
www.r

In a similar way as in the original study, the marginal effect of each activity, accounting for concurrent activities and subject effects through mixed effects models, is presented in Table 4. The activity with strongest effect on wastage of time in the replicated study was again performing Additional code analysis, associated with a waste increase of 10.4 percentage units (p.u.) (95% CI 3.9 to 18.2 p.u.), followed by Additional searching for documents (mean waste increase 8.7 p.u) and Additional Refactoring (mean waste increase 8.6 p.u.).

Table 4
The average effect of TD activities on wasted time from the replicated study.

What extra activities was the wasted time spent on?	Estimated effect on waste increase* (95% CI)**
Additional code analysis	10.4 (3.9; 18.2)
Additional searching for documents	8.7 (−0.0; 17.1)
Additional refactoring	8.6 (1.6; 15.8)
Additional work-arounds	6.9 (−3.2; 15.0)
Additional communication	1.8 (−4.1; 9.2)
Additional testing	1.8 (−3.8; 8.9)

* Estimated effects are presented in percentage units.
** 95% confidence intervals were computed using the non-parametric bootstrap percentile method, using 10,000 bootstrap replicates.

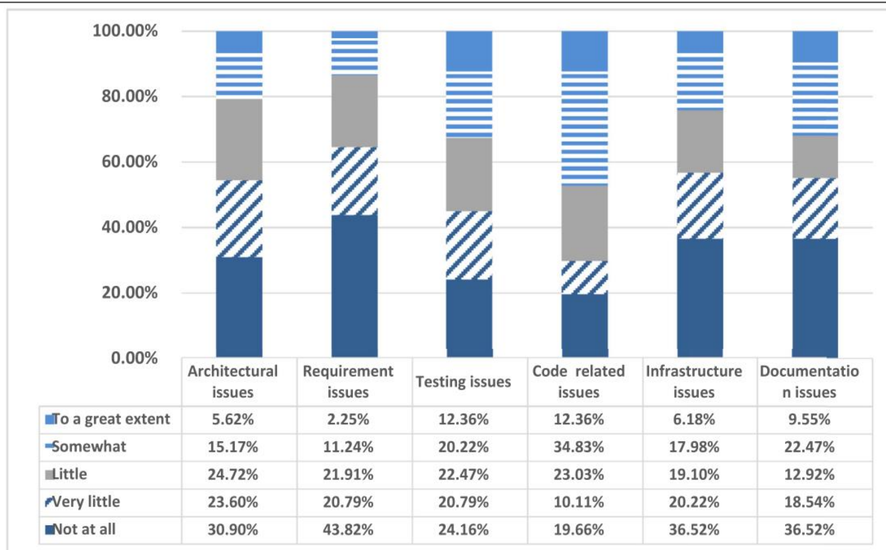
To conclude, the original and the replicated studies show a somewhat different relationship and ranking between the amounts of wasted time in relation to the different activities, even if the mean value of each individual activity is quite similar to the values in the original study.

13.5.7.3. Replication of result addressing technical debt types

When answering research question RQ3, we examined the extent to which different TDs were encountered in order to understand their frequency.

المنارة للاستشارات
www.m

Table 5
The Likert scale of each encountered TD type.



	Architectural issues	Requirement issues	Testing issues	Code related issues	Infrastructure issues	Documentation issues
To a great extent	5.62%	2.25%	12.36%	12.36%	6.18%	9.55%
Somewhat	15.17%	11.24%	20.22%	34.83%	17.98%	22.47%
Little	24.72%	21.91%	22.47%	23.03%	19.10%	12.92%
Very little	23.60%	20.79%	20.79%	10.11%	20.22%	18.54%
Not at all	30.90%	43.82%	24.16%	19.66%	36.52%	36.52%

As shown in Table 5, a significant proportion of the encountered TD is related to source code and testing, where 12.36% of the TD for those types is encountered ''To a great extent.''

Even though the two sets of results are not identical, they share the same finding when studying which TD types are most often and most seldom encountered to a large extent. When comparing the result from the original study with the result from the replicated study, it is apparent that the code-related TD and test related TD are the two TD types that are most often encountered "To a great extent." Furthermore, the results in both studies show that requirement related TD is most seldom encountered "To a great extent."

To conclude, these replicated results confirm previous findings and contribute additional evidence that developers suffer from several different TD types and that they differ in terms of their frequency and magnitude.

13.6. Discussion

The following subsections present discussions and limitations for the research results presented in Section 5, and the results are grouped according to each research question followed by a section addressing the replicated phase of the study and, finally, a section about the implications for practitioners and researchers.

13.6.1. Wasted time and introduction of new TD

The first three research questions (RQ1, RQ1.1, and RQ1.2) focus on how much software development time developers are wasting due to TD, and the fourth question (RQ4)

المنارة للاستشارات

www.m

addresses to what extent developers are forced to introduce new TD because of already existing TD.

The most striking finding shows that developers waste almost a quarter of all development time due to TD, and, even if different patterns of the distribution over calendar time were observed, the overall distribution of the wasted time did not show any clear trend.

Even if this study does not explore if, and in what ways, the first introduction of TD affected the productivity of the development work, this result indicates that the present TD causes a great deal of wasted time during the overall development work, where the ratio of development effort and maintenance effort in a product development lifecycle becomes tilted toward more maintenance effort due to the presence of TD in the software.

Moreover, this indicates that, if the software companies are not aware of this time and have not calculated for it, they could easily end up with time pressure, forcing them to introduce additional TD. In fact, the developers report that a quarter of all encountered TD forces them to introduce additional TD. This result indicates that, depending on software companies' degree of ability to remediate TD, the amount of TD could potentially increase continuously, and, in the worst case, this could lead to a vicious circle of TD growth. This result quantitatively corroborates the findings of [163], where the term contagious debt is described.

13.6.2. Additional activities

The second research question (RQ2) in this study sought to explore which activities the wasted time was spent and also whether the amount of the wasted time was related to any specific activity. The result shows that the most common activity on which the extra time was spent in performing additional testing, followed by additional source code analysis and additional refactoring.

This result implies that, if the systems did not have TD, the time spent on these activities could be reduced. Furthermore, spending a great deal of time on these activities during the software development could, consequently, potentially be an indicator of a system suffering from TD and also an indicator of the amount of interest that has to be paid and thereby indicate a decrease in developer productivity.

13.6.3. Technical debt types

The results from the third research question (RQ3) show that all TD types are significant and strongly associated with the amount of the wasted time, whereby *Source code* TD has the strongest association with the amount of wasted time. This result demonstrates that all the different types of TD require attention. A possible explanation for these results may be that developers have a higher awareness of Source Code TD and, therefore, experience its negative impact as more prominent. Likewise, the results show that developers encounter less Requirement TD and Infrastructure TD, which also points to the idea that developers are less prone to attribute the wasted time to those TD types.

المنارة للاستشارات

www.r

This result further implies that software companies need to focus on several different types of TD and not, as is currently the case, focus primarily on code-related TD.

13.6.4. Awareness and challenges

The fifth and sixth research questions (RQ5 and RQ6) address the levels of awareness of the developers and their manager regarding the amount of time wasted due to TD, the benefits of this insight, and how they communicate these issues within their organizations. From the results, we can see that software developers are reasonably aware of the amount of time they waste during the development phase, despite the fact that they do not attempt to measure, track, or quantify it.

However, the managers of the developers have a much lower awareness of the amount of time the developers waste, and the professions also seem to have different views on what is a reasonable or unreasonable amount of time to waste on TD. Both developers and managers could see the benefits of quantifying the amount of wasted time, but both professions were, in general, reluctant to practically implement a systematic approach to quantifying the wasted time. Developers argued that reporting the wasted time due to TD would require additional time and effort, while the managers were hesitant to implement such measures due to the extra workload it would place on the developers. If the managers are not aware of the amount of software development time the developers waste because of TD, they are consequently not able to react and take appropriate action regarding the wasted time. This means that, in a worst-case scenario, the amount of wasted time and the lack of developer productivity could end up being increased instead of being reduced. In the long run, low developer productivity can, due to time pressure, stress the developers to further introduce new TD and can also have a negative impact on the amount of new features that can be implemented and can harm both the maintainability and evolvability of the software product.

13.6.5. Replication study

In general, a successful replication of a study is one that helps the research community gain information about conditions under which the results hold [71], [72].

The replication phase of this study aimed at replicating the results addressing RQ1.1, RQ2, and RQ3. The replicated results for RQ1.1 and RQ3 where, overall, in line with the results from the original study, which implies greater reliability and confidence in these results.

Further, the results for RQ2 contradicted to some extent the results from the original study addressing the different activities on which the wasted working time was spent.

A hypothetical explanation of this discrepancy may be found in how the data was collected in the replicated study. One of the major alterations of the data collection in the replicated study is that it was collected at only one company (compared with six companies in the original study). This replication setting creates a context in which the

المنارة للاستشارات

www.m

sources of variations potentially could be limited. One could expect that several of the participating developers in the replication study worked in a similar environment or even in the same code base, thus experiencing TD that required specific activities to take place, resulting in a company-specific result that is less generalizable to a broader community.

However, even if the results from this research questions show a different ranking of how strong each activity is associated with the amount of wasted time, the mean value of each activity was quite similar to the original results. In the replication study, the result showed that performing additional code analysis had the strongest association with the amount of wasted working time due to experiencing TD, as compared with the original study where performing "Additional Testing" had the strongest association. This result could potentially be explained by a company-specific environmental setting, but further studies that take these variables into account will need to be undertaken.

13.6.6. Implications for practitioners and researchers

It is commonly quite difficult to motivate and argue for the need to prioritize refactoring activities due to software experiencing TD in today's software industry. One major reason for this can be described in terms of the lack of knowledge about how TD negatively affects the software developer productivity.

This study has shown that an extensive amount of valuable working time is wasted due to TD and that this TD causes the developer to perform different activities that would not have been necessary if the TD were not present.

However, being able to describe and understand the amount of the negative effects of TD in terms of wasted time can help when developers argue for the need to initiate refactoring to reduce the amount of TD and thereby potentially decrease the future amount of time wastage.

This study makes a novel contribution to the existing body of knowledge and suggests several important practical implications that demonstrate the impact TD has on software development productivity. This contribution of this study can be used by both software practitioners and researchers within the field:

- Based on this study's empirical result, we show that software developers report that they waste, on average, 23% of their working time due to TD.
- We present results showing that the wasted time is most commonly spent on performing additional testing, followed by conducting additional source code analysis and performing additional refactoring.
- The results show that in almost a quarter of all occasions when encountering TD, the developers are forced to introduce additional TD due to the already existing TD.
- This study provides new insights into TD research by revealing that the developers are largely aware of the amount of time they waste due to TD.

المنارة للاستشارات

www.r

However, the study shows that the developers' managers are not as aware of the amount of time developers waste and that the different professions seem to have different views on what is a reasonable or unreasonable amount of time to waste due to TD.

- This study shows that none of the companies tracked or measured the amount of wasted time due to TD, and none of the companies had an aligned strategy for addressing the interest of TD.
- This study shows that both developers and managers see the benefits of tracking the amount of wasted time, but both professions are somewhat reluctant to implement such measures in practice. This unwillingness is recognized as a challenge for companies.
- We provide an empirically based study on how TD negatively affects practitioners within the software industry, based on both quantitative and qualitative data. A major strength of this study is the longitudinal research, which increases the validity of the results compared to cross-sectional studies.
- Overall, these findings suggest strong recommendations for software companies to focus further on continuously undertaking refactoring initiatives of TD issues to keep the amount of TD at bay on an ongoing basis. In general terms, this means that such TD remediation and prevention initiatives also would have a positive impact on the overall developer productivity.

13.7. Verifiability, limitations, and threats to validity

The purpose of this section is to reflect on the extent to which this study has addressed the goal of ensuring verifiability, describing limitations, and finally addressing potential threats to validity.

13.7.1. Verifiability and limitations

There are several important limitations that necessitate a cautious interpretation of the results of the present study. First, selection bias is a potential limitation since the data from the invited companies in the study was gathered only in specifically chosen companies. Second, given the self-reported nature of the collected data in the surveys, the findings should be interpreted with caution, particularly because, during the longitudinal data collection phase, the surveyed developers may have had insufficient knowledge and ability to categorize and quantify the correct TD type and to quantify the correct amount of wasted time due to TD.

However, one could argue, since reporting the time spent on different tasks and activities is a common practice for developers performing their time registration, their ability to report the time should be reasonably sound. With this as a background, together with the provided education material, guiding the participating developer to distinguish between

المنارة للاستشارات

www.r

different types of TD would assist in determining the amount of wasted time and the specific type of TD on which it is being wasted.

Third, a note of caution is due, since this study's result is derived from reports from *developers* and *managers* only, meaning that the findings cannot be generalized to other software practitioner roles.

13.7.2. Threats to validity

The result of this study may be affected by some threats to validity such as internal validity, external validity, construct validity, and reliability.

The major threat to the internal validity of this research design is when the causal relationships between the wasted time and the different TD types and the different activities were examined, as it affects our ability to explain accurately the phenomena that we observed [66]. To mitigate this threat, we have adopted both a univariable and a multivariable analysis of the data.

In this work, we have analyzed data to find a correlation between specific parameters in the reported data. However, we do acknowledge that this correlation does not imply causality between the variables and that the same results may not be reached if considering another collection of companies or developers with different characteristics. However, to further mitigate this threat, we conducted follow-up interviews with 12 of the participating developers in which the relationship between the reported amount of wasted time and the listed additional activities and the different TD types were assessed. Several of the findings were also validated by an additional replication study using a different and independent data sets, concluding and strengthening several of the derived results.

Furthermore, to mitigate the potential threat to the validity of self-reports, all participants reported the wasted time related to TD for a short period of time (on average 3.1 days). The confidence of self-reporting data was also supported by the fact that the practitioners knew that the surveys were coming, so they could pay special attention to their working tasks and effort spent. In addition, in management research, it is not uncommon to use self-assessment when studying participants' productivity since this method is considered as a consistent method for objective measurements of performance [172].

The external aspect of validity addresses the extent to which it is possible to generalize the findings [73]. The responses that our respondents gave might not be representative of the entire developer population. Although we cannot generalize the results, we can rely on a relatively high number of participating organizations (6), working in different business and application domains. Furthermore, in surveys, there is always a risk that the sample is biased, and, therefore, a potential threat relates to, for instance, the geographical, cultural, and demographic distribution of response samples. However, to confirm the generalizability of this study and to mitigate this threat, we have replicated

المنارة للاستشارات

www.r

the study with a different set of respondents, both in terms of geographical area and software development culture.

Construct validity addresses the extent to which the operational measures that are studied accurately represent what the researchers are considering [73]. This threat is related to whether we can correctly use the amount of wasted time as a substitute for software development productivity and whether the data collection approach is well-designed for the research purpose.

Using only a single report for each respondent involves a risk that this reporting gives a measurement bias [86]. To mitigate this threat, the data were collected using several reporting occasions over time, using a longitudinal data collection approach. This approach reduces the subjectivity of only studying the reported data on one single occasion.

Furthermore, this threat also relates to whether the study constructs are defined and interpreted correctly by the respondents [73]. To mitigate this risk and to ensure that the respondents had the same base of knowledge in the field of the study, all participants in the longitudinal study received the educational material before starting the study. Another threat concerning the survey questions relates to whether the question could be clearly understood by the participants. To mitigate this threat, the initial survey draft was reviewed by all the authors, and we additionally made a pilot study with one software practitioner to examine the understanding of the survey questions.

The goal of reliability is to minimize the errors and biases in a study [66]. Reliability addresses whether the study would yield the same results if other researchers replicated them, following the same procedure, by means of the extent to which the analysis is dependent on specific researchers [73], [66].

To mitigate this threat in the original study, following guidelines by Yin [66], we designed the study in six distinct and separately documented phases (see Section 4.1), and made these steps as operational as possible to assist the repeatability of the study results.

As mentioned briefly in the above section, this study also includes an additional phase with the goal of replicating several of the findings. This replication study also assisted in mitigating several validity issues of the original study. In terms of external validity, the replication of the study assisted us by showing that several of the original results were not dependent on the specific conditions of the original study. The independence of the replicators from the original study lends additional confidence that the original results were not the result of data collection bias.

Similarly, in terms of internal validity, the replication study also assisted in showing the range of conditions under which the results hold. Since the several variables of the replication study were different from those of the original study (e.g., a new set of respondents, different data collection design, etc.), the replication phase contributed some confidence that the findings are not limited to the particular setting we had in the original

المنارة للاستشارات

www.r

study. Consequently, the additional replication study addressed both external as well as internal validity.

13.8. Conclusion and future work

This study set out to analyze the negative effect TD has on software productivity from the point of view of software developers and their managers.

This study reports on the replication and extension of a longitudinal study of technical debt, where 43 developers reported twice a week for seven weeks how much time they waste due to TD, on which additional activities this time was spent, and what type of TD caused the wasted time.

This study provides evidence that TD hinders software developers by causing a substantial amount of wasted time. This wasted time negatively affects the development productivity and viability of the software. Even if both developers and their managers clearly see the benefits of reporting the wasted time, it is a challenge to implement such a reporting task due to unwillingness and time restrictions. This study shows that TD also contributes to the need to perform time-consuming additional activities, and developers report that, on average, 23% of all software development working time is wasted due to TD.

Furthermore, due to the presence of TD during the development work, developers most commonly have to perform additional testing, source code analysis, and refactoring. This study also shows that, in a quarter of the occasions where developers encounter TD, they are forced to introduce additional TD due to the already existing TD. This burden of being forced to introduce additional TD demonstrates the contagiousness of TD, and our results suggest that TD should be prioritized for refactoring because it forces the developers to introduce further additional TD, which generates even more interest.

These findings indicate that software companies need to be armed with strategies and proactive management to enable them to track the interest of TD. Such a strategy could result in better, more informed decisions to balance the accumulation and the repayment of TD.

It was not possible in the present study to study the relationship between the qualities of the developers' software in relation to the wastage of their working time. However, as part of future work, we plan to extend this study by applying a triangulation of the quantum of TD within the investigated software system by, for instance, using tools for source code statistics, test statistics, or code churn metrics. This extension and replication of the study would provide additional heterogeneity in the relationship between TD and the productivity loss over different values of TD.

المنارة للاستشارات
www.r

ACKNOWLEDGMENT

Many thanks to the industrial partners who participated in both the original and the replication study and interviews. We would also like to thank Henrik Imberg for his valued support during the statistical analysis of the data.

المنارة للاستشارات

www.r

للاستشارات المنارة

www.m

14. The Influence of Technical Debt on Software Developer Morale

This chapter aims to explore how software developers' morale is influenced by TD and how their morale is influenced by TD management activities. Furthermore, the study presented in this chapter also correlates the morale with the amount of wastage of time due to TD.

Previous research in the Technical Debt (TD) field has mainly focused on the technical and economic aspects, while its human aspect has received minimal attention.

Firstly, we conducted 15 interviews with professionals, and, secondly, these data were complemented with a survey. Thirdly, we collected 473 data points from 43 developers reporting their amount of wasted time. The collected data were analyzed using both quantitative and qualitative techniques, including thematic and statistical analysis. Our results show that the occurrence of TD is associated with a lack of progress and waste of time. This might have a negative influence on developers' morale.

Further, management of TD seems to have a positive influence on developers' morale. The results highlight the effects TD has on practitioners working life. This study presents results indicating that software suffering from TD reduces developers' morale and thereby also their productivity. However, our results also indicate that TD management increases developers' morale and developer productivity.

This chapter has been published as:
The influence of Technical Debt on software developer morale,
T. Besker, H. Ghanbari, A. Martini, and J. Bosch,
Journal of Systems and Software, vol. 167, pp. 110586, 2020.

14.1. Introduction

In recent years, there has been an increasing interest in technical debt (TD) within the software engineering discipline. The majority of previous studies focused on investigating the technical, financial, and organizational aspects of TD [173], [15],[153]. However, until now, the literature has paid minimal attention to the human, and social aspects of TD, including the role of developers in the occurrence of TD as well as its consequences for them. More recently, several studies have suggested that TD has a negative influence on developers' emotions and affects [10],[11],[12] and their morale [7],[13], where morale can be explained as a multidimensional concept that "*subsumes confidence, optimism, enthusiasm, and loyalty as well as a sense of common purpose*" [174]. However, none of these previous studies specifically focuses on *empirically* investigating and explaining the relationship between TD, the developers' morale, and software developer productivity. Since no known research has focused on exploring these relationships empirically, this study seeks to obtain data from practitioners who experience TD on a daily basis, in order to expand the empirical findings in this area.

المنارة للاستشارات

www.r

In a study by Tom et al. [7], the authors suggest that morale, alongside quality, productivity, and project risk, are the four main areas that are negatively influenced by the occurrence of TD. The authors claim that, since the occurrence of TD reduces software quality, developers must spend more time and effort to address quality issues in the future. This, in turn, will decrease developers' productivity and maintenance costs in the long term [7],[13]. On the other hand, some studies suggest that developers do not feel comfortable about taking on TD [10],[11],and it lowers their motivation [12].

Software development is a sociotechnical phenomenon [175], and therefore, its success depends on both its social and technical aspects [176]. The ways in which today's software developers work require a comprehensive understanding of their feelings, perceptions, motivations, and identification with their tasks in their respective project environments [177]. Recent studies have shown that positive affective states, such as happiness, satisfaction, and motivation, increase software developers' productivity and software quality [176],[178]. On the other hand, based on the results of previous studies, mainly from management science [179],[180] and more recently in software engineering [181],[182],[[183],[184], developers' morale and their productivity seem to correlate. However, to the best of our knowledge, there is a lack of studies focusing on investigating the relationship between TD and morale and productivity using empirical data.

Considering the large number of previous studies which argue that software firms take on TD to boost their productivity and obtain added business value [7], it becomes apparent that minimal attention has been paid to the human and social aspects of TD [185], and that there is a need for empirical studies [186] to explore and explain the effects of TD on developers' morale and to correlate it with developer productivity.

The goal of this study is, therefore, to understand how software developers' morale is influenced by TD present in the software they are developing and also to understand how their morale is influenced by TD management activities. Furthermore, in order to understand if their morale affects their work productivity, this study explores associations between morale with the amount of wastage of working time due to experiencing TD.

Note that the occurrence of TD and its produced time waste are two separate concepts: companies can have TD with a low "interest rate" (the term used to characterize the negative impact of TD). In such case, the presence of TD does not produce time waste but can still influence morale. On the other hand, TD can produce time waste, which may be the reason why TD affects morale. For these reasons, we study the concepts separately and we have two separate research questions (RQ1 and RQ3).

The term developer morale will be examined by surveying developers to indicate on a Likert scale their opinion about the impact of TD on three different dimensions of morale. The three different dimensions are: a) *Affective antecedents* such as focusing on developers' moods, feelings, emotions, and attitudes, and b) *Future/goal antecedents* with a focus on the developers' goals for the future and, finally, c) *Interpersonal antecedents* addressing the relationships and communication between developers. These dimensions are described in detail in section 2.2.

Based on this goal, this study will examine the following research questions (RQ):

المنارة للاستشارات

www.r

- **RQ1: Does the occurrence of TD influence developers' morale?**
- **RQ2: Does the management of TD influence developers' morale?**
- **RQ3: Does wasted time (due to TD) correlate with morale?**
- **RQ3.1: Does wasted time correlate with TD occurrence dimensions of morale?**
- **RQ3.2: Does wasted time correlate with TD Management dimensions of morale?**

Our study has several novel contributions to software engineering research and practice. First, our study specifically concentrates on investigating the influence of TD on developers' morale. Our findings are encouraging since they clarify the impacts of TD on different dimensions of morale and illustrates its relationship with developer productivity. Especially by indicating that developers consider TD and its management as important factors influencing progress and future development activities, this study encourages software firms to consider the human and organizational consequences of TD more seriously. Additionally, since previous studies have indicated the link between morale and developers' productivity, our study investigates this relation empirically. In future research, this approach will enable enhanced exploration and measurement of software developers' morale in different contexts.

An earlier paper at the 11th International Symposium on Empirical Engineering and Measurement (ESEM) [187] presented part of the results reported in this study. However, this manuscript extends that previously published study significantly by triangulating the previous data collection and adding more data and, moreover, by adding more analysis. The novelty of this study's approach compared to the previous study lies in the morale focus of this paper, where we correlate the waste of time with three different dimensions of morale together with seven additional interviews and additional survey questions related to the morale perspective of TD.

The data representing the wasted time are based on reported data from developers who were asked to individually keep track of the time they wasted due to experiencing TD, on a daily basis. They were thereafter asked to themselves report this time to us, twice a week in a survey. Meaning that the data are based on the participants' own tracking and calculations.

The original study was exploratory, and based on our initial results, we proposed a set of propositions. In this study, using those propositions as a starting point, we conducted an additional cycle of research to investigate further the validity of those propositions and also to explain the logic behind the correlation of TD, morale, and productivity. In order to investigate if developers' morale affects their productivity, we have added one additional research question (RQ3), where we correlate the reported amount of working time wasted due to experiencing TD (as a proxy of productivity) with three different dimensions of morale and discuss them in detail. The related Research section has been extended to be broader and more carefully cover additional related research publications. We have extended the data collection by adding a longitudinal study collecting additional

المنارة للاستشارات

www.m

quantitative data in order to understand how much time developers report as being wasted due to experiencing TD over a longer period of time. We also augment our previous study with seven additional interviews. We believe that this additional context extends and strengthens the results derived in the original study.

This paper is structured as follows: Section 2 presents the theoretical background of the study. In section 3, we describe the research methodology. Section 4 presents the research results. Sections 5 and 6 discuss the findings and threats to the validity of the study, respectively. Finally, Section 7 concludes the study.

14.2. Theoretical background

In this section, we discuss the background of the study in terms of TD, morale, the relationship between TD and morale, and the relationship between TD and productivity, and finally, the relationship between morale and productivity.

14.2.1. Technical Debt

In recent years, TD has been widely studied in the software engineering literature [15],[186],[188]. TD is composed of a debt, which is a sub-optimal technical solution that leads to a short-term benefit as well as to the future payment of interest, which is the extra cost due to the presence of TD (e.g., slow feature development or low quality) [153]. The principal is regarded as the cost of refactoring TD. Although accumulated TD might result in a gain between the short-term benefit and the interest paid, in many cases, documented in the literature and software projects, the interest might largely surpass the gain by, for example, leading to development crises [110].

There are several kinds of TD, such as Architecture TD, Testing TD, and Source Code TD [15], depending on where the sub-optimality has occurred, what artifact, and what level of abstraction. In recent years, there has been an increasing interest among software engineering researchers in providing novel solutions enabling software developers to manage TD. Management of TD consists of a set of activities that enable software development teams to both prevent the occurrence of TD and deal with existing TD and its unwanted consequences[15]. It is worth noting that, in their systematic mapping study, Li et al. [15] classify TD management activities into identification, measurement, prioritization, prevention, monitoring, repayment, documentation and representation, and communication. However, discussing each of these activities is out of the scope of this study.

14.2.2. Human factors related to Technical Debt

Today's software development work is, to a great extent, a human-based activity that depends on several different human aspects [189] whereby its success is dependent on financial, social, physical, and emotional factors [190].

المنارة للاستشارات

www.r

The success of software development projects is dependent on the practitioners working with the software. For instance, Beecham [191] states that developers' motivation is an important issue, and that lack of motivation is one frequent cause of software development project failure [191]. Furthermore, Verner et al. [192] state that motivation is considered to be the single largest factor in developer productivity.

Moreover, several researchers state that, specifically, TD has a negative impact on the practitioners' daily work with software that suffers from experiencing TD. Tom et al. [7] state that it is likely that developers find the effects of TD frustrating in the long term, and Laribee [193] explains that economic downside, there’s a real psychological cost to technical debt that causes both frustration and a sense of helplessness to the developers and also Vogel-Heuser and Neumann [194] state that human factors can be a reason for TD.

Further, software engineering success is described by [195] to increasingly dependent on the well-being of developers’ communities, and Tamburri et al. [21] highlight that a suboptimal development community can have a significant impact on the software by causing unforeseen project cost. The authors refer to this suboptimality as social debt. Alfayez et al. [196] reveal that the skills and habits of developers have an impact on the software where e.g., developer commit frequency and seniority have a significant negative relationship with the TD introduced by the developer, meanwhile, Salamea and Farré [197] conclude that communication skills have barely any impact on TD.

14.2.3. Morale

Morale, as a research topic, has originally been of interest within the military discipline. However, after the Second World War, it has received increasing attention from other disciplines such as organizational sciences, management, education, and healthcare [198]. Despite its vast literature, morale lacks a coherent and precise definition, which increases the risk of researchers measuring other concepts instead of morale. For example, in an extensive literature review, Hardy [198] shows that several concepts such as satisfaction, motivation, and happiness, have been used interchangeably to discuss morale. Similarly, França et al. [199], describe that the terms morale, inspiration, enthusiasm, and motivation are popular synonyms even if they potentially have distinct actual meanings in the field of psychology.
However, by conducting a longitudinal empirical study, Hardy [198] clarifies that morale is different from these concepts. This study will only focus on morale as a human factor, where we use the definition of morale provided by Peterson et al. [174] as *“a cognitive, emotional, and motivational stance toward the goals and tasks of a group. It subsumes confidence, optimism, enthusiasm, and loyalty as well as a sense of common purpose*.” As can be understood from this definition, morale is a multidimensional concept consisting of different affective, interpersonal, and motivational dimensions. Therefore, it becomes obvious that for studying morale, a combination of unidimensional concepts such as satisfaction, motivation, or happiness must be considered.

Previous studies have used different qualitative and quantitative methods for measuring morale, including direct (e.g., [200],[201]), and indirect measurement methods (e.g.,

المنارة للاستشارات
www.m

[110]). In its simplest way, morale has been widely measured directly by using single-scale measures in which individuals are asked to rate their level of morale [200]. However, using this approach becomes problematic since morale is a subjective phenomenon, and therefore, there is a danger that respondents have different perceptions of morale [198]. Consequently, Hardy [198] suggests an approach for predicting the levels of morale from measuring a set of factors that influence morale. These antecedent factors of morale are divided into three main categories.

Affective antecedents consist of factors related to moods, feelings, emotions, and attitudes. For instance, high morale is often associated with positive affective states (e.g., sense of achievement, recognition, or happiness), while low morale is associated with negative affective states (e.g., feelings of failure, criticism, or injustice) [198]. The second category, *future/goal antecedents*, consists of factors related to the perception of individuals about the difference between the future and today as well as their goals for the future. For instance, believing in a better future, sense of progress, or success are considered as being key factors in increasing morale while lack of clarity, confidence, or progress are considered to lower morale [198]. Finally, the *interpersonal antecedents* of morale pertain to the factors related to the relationships and communication between individuals. For instance, having good relations with others, teamwork, or helping each other contributes to higher levels of morale while a bad work atmosphere or perceiving bullying or being dragged down by others can reduce morale [198].

14.2.4. The Relationship between Technical Debt and Morale

The morale of software practitioners can be affected by several different things, such as, for example, the maturity or stability of the adopted development process [202] or by the quality of the overall software product architecture [203]. However, this study will only focus on the impact that TD has on morale.

Tom et al. [7] argue that TD has a very strong negative effect on developers' morale. Following this argument, we tried to identify previous studies that discuss the relationship between TD and morale. In so doing, we used Google Scholar to search for studies mentioning both Technical Debt and morale. The search, which was conducted in January 2020, returned 415 potentially relevant results. After reading these results and checking their list of references, we were able to find only twelve sources mentioning the potential influence of TD on developers' morale or emotions (see Table I).

TABLE I - PREVIOUS STUDIES ARE MENTIONING THE RELATIONSHIP BETWEEN TD AND DEVELOPERS' MORALE.

Ref	Argument	Source of Argument

المنارة للاستشارات

www.r

[11]	TD will hurt you later. It is better to pay upfront.	An Interviewee's opinion
[13]	Uncontrolled technical debt has a negative impact on the developers' morale.	Literature, Tom et al. [7]
[8]	Engineers don't like technical debt because they want to create perfect software.	An Interviewee's opinion
[200]	Working off debt can be motivational and good for team morale.	Author's opinion
[201]	TD reduces developers' motivation.	Author's opinion
[12]	Working off debt can be motivational and good for team morale.	Literature [200]
[7]	TD ultimately has negative impacts on morale.	Author's conclusion
	Developers would find the effects of technical debt frustrating in the long term.	Author's interpretation of a blog-post and several interviewees' opinions
	A lot of the tasks to prevent or repay the accrual of TD aren't very fun.	Several interviewees' opinion
[10]	Taking TD doesn't feel good for developers.	An interviewee's opinion
[204]	Apart from technical challenges, technical debt also impacts the morale and motivation of the developer team.	Author's conclusion

www.r

Evans Data Corp, CIA Factbook, and Stripe research [205]	76% of the developers reported that paying down TD hurts their morale.	Result from a survey in a online- report
[206]	Messy code will drive messy behavior and frustration, limit understandability, and, consequently, induce stress in its developers and invoke a vicious cycle of bugs, vulnerabilities, and technical debt that is hard to get past. This cycle will eventually drive developer morale down.	Author's conclusion
[207]	The study finds that the negative impacts of TD in startups would be on morale, productivity, and product quality. -> Refers to Greenfield study	Literature Giardino et al. [6]

We found three articles and one blog post that have mentioned the influence of TD on developers' morale. Also, during our literature search, we found four articles that mention the influence of TD on developers' emotions. However, analyzing these sources revealed that only four of them support their arguments with empirical evidence, and the rest either cite other sources or report opinions. In addition, all these sources only mention the relationship between TD and developers' morale or emotions, and investigating this phenomenon is not their primary research focus.

Taken together, we aim to understand how the occurrence of TD and its management may affect the three dimensions of morale suggested by Hardy [198]. Therefore, we decided to conduct an exploratory field study to understand how TD and possibly its management could potentially influence software developers' morale. To achieve this, we had to adopt a research approach where we utilize the indirect approach suggested by Hardy [198] to explore the potential influence of TD and its management on developers' morale (see Fig. 1). Using this approach not only enables us to investigate the impacts of TD and its management on developers' morale but also to gain a better understanding of the potential consequences of TD for developers and understand which dimensions of morale are more influenced by TD.

14.2.5. The Relationship between Technical Debt and Developer Productivity

Software developers' performance has a direct impact on software development productivity [208], and even if the term "waste of time" is commonly used within Lean organizations to target increased productivity, there are only a few studies looking into waste in software development organizations [209]. As previously mentioned, several

المنارة للاستشارات
www.n

researchers describe that software suffering from TD has a negative impact on software development productivity. For instance, Graziotin et al. [210] state that affective states such as emotions, moods, and feelings have an impact on the productivity of individuals.

There is no commonly agreed definition of productivity [43], but in software engineering, productivity is commonly defined, from a financial perspective, as the effectiveness of productive effort measured as the rate of output per unit of input [43], [44], [45],[30]. Productivity is also a measure of the quality of an output relative to the input required to produce the output. This means that productivity is a combined measurement of efficiency and quality. However, [43] argue that "*there is no metric that adequately captures the full space of developer productivity*" and instead, encourage the design of a set of metrics tailored for answering a specific goal and a purpose-based definition of the desired software value.

Several constraints can influence software development productivity, such as cost, schedule, and scope [161]. Moreover, Oliveira et al. [45] express that researchers have not yet reached a consensus on how to measure productivity properly in software engineering. However, in this study, we have decided to focus on productivity in terms of the amount of wasted software development time due to developers being impeded during their daily software development work by experiencing TD within their software. However, we do acknowledge that lack of waste does not, by default, guarantee productivity. Our intention is not to redefine the term productivity to software practitioners and organizations, but we motivate this proxy based on the reason that if less time were spent on "extra" work tasks and activities due to experiencing TD, more time could be spent on other activities that would increase the overall productivity.

Sedano, Ralph, and Péraire [211] echo this notion by stating that "*waste is any activity that produces no value for the customer or user*," and reducing this waste of time would thereby improve the software development productivity. In their study, they identified that TD could decrease the developer's productivity both in terms of wasted time and by causing reworking and an extraneous cognitive load.

This paper is somewhat related to a previous study we conducted [212]. In that study, we more specifically studied the amount of wasted working time due to TD. The result of that study shows that TD contributes to the need to perform time-consuming additional activities and that developers waste, on average, 23% of all developers working time due to TD.

14.2.6. The Relationship between Morale and Productivity

Previous research suggests that the level of employees' morale is correlated with their productivity [181], [182], [183], [198], and project success [184]. This correlation has been explained mainly in terms of employees' perception of their progress [182],[184],[198],[191], and their personal accomplishments [181],[183],[213].

Thus, employees' progress is explained mainly in terms of their perception about the amount of work that is completed and the amount of unnecessary waste [182],[184],[198]. For instance, empirical data collected by Hardy [198] suggests that individuals who

المنارة للاستشارات
www.r

perceive their morale too low consider the time spent on performing their work is less than the time they waste on organizing the workload. Fairley and Willshire [182] suggest that software firms can significantly increase developers' morale and productivity by eliminating avoidable reworks (i.e., waste). A sufficient amount of rework (e.g., refactoring or testing) is known to be beneficial in software projects. For instance, Damian and Chisan [181] suggest that developers' pride achieved as a result of continuous software improvements has a positive influence on developers' morale and ultimately leads to higher productivity. However, Fairley and Willshire [182] argue that a majority of software firms deal with a significant amount of unnecessary reworks, which could have been avoided if the software development tasks had been performed correctly in the first place (e.g., by avoiding the unnecessary accumulation of TD). McConnell [200] also suggests that such avoidable reworks are very costly and lead to a waste of development and maintenance time. Previous research suggests that TD taken reactively as a short-term solution such as minimizing documentation or testing or "dubious budget savings" [200] lowers both morale and productivity [183].

To conclude, several researchers have pointed towards the idea that working with software suffering from TD has an impact on the developers' morale and further that morale influences objective productivity indirectly. However, to the best of our knowledge, there is a lack of empirical studies assessing the relationships between TD, morale, and productivity.

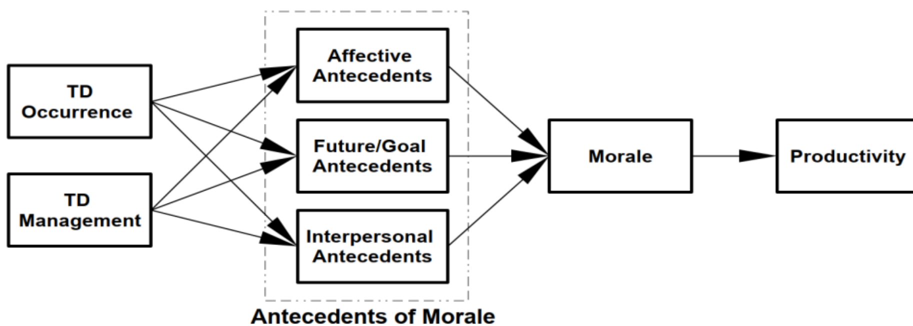


Fig. 1. The conceptual framework

The conceptual framework is shown in Fig. 1 and summarizes the above discussion. As illustrated in this figure, in this study, we aim to investigate the relationship between TD and developers' morale and its influence on their productivity. In particular, we try to understand how the occurrence and management of TD could affect the three dimensions of morale discussed above. Using this approach not only enables us to capture the attitudes of developers towards TD and its management but also to understand which dimensions of morale are more influenced by TD and its management. Additionally, we try to capture any potential relationship between developers' morale and their productivity in terms of the amount of time they waste due to the occurrence of TD. To study the relationships shown in our conceptual model, we have conducted an empirical study using a mixed-methods approach.

المنارة للاستشارات

www.r

14.3. Methodology

There are different types of empirical studies. Since our goal is to study TD and morale in its natural context, we have adopted a mixed-method approach where we use both qualitative and quantitative data collection and analysis methods, and part of the study also collects data longitudinally.

This study is based on data from 15 face-to-face interviews, and a survey, together with a longitudinal study, in order to examine the negative impact TD has on software developer morale and developer productivity. The study was conducted between September 2016 and January 2018.

As visualized in Fig 2, the overall research design was divided into six phases. The figure represents both activities that were performed in the initial study (darkest grayed boxes), the activities that were conducted partly in the initial study but enhanced in this extension of the study (light grey boxes), and also the additional research activities that are new in this part of the study (white boxes). The first four phases have a focus on answering RQ1 and RQ2, and when answering RQ3, all six phases are involved. Meaning that in phase 5 and 6, we collected data that we used together with the data from the previous phases when answering RQ3. The following sections describe each phase and the related research methods used in each stage.

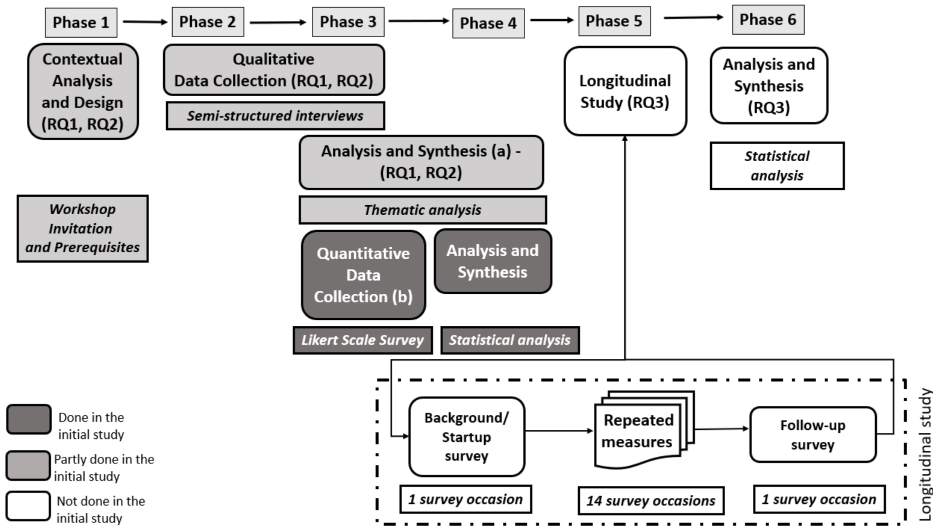


Fig. 2. Visualization of the research design and research method used in each phase

14.3.1. Phase 1—Contextual Analysis and Design.

First, the study was presented and discussed during a workshop with software practitioners from several software companies, all having an extensive range of software

المنارة للاستشارات
www.r

development. The selection of companies was carried out with a convenience sample of industrial partners within our network. This phase acted as a guide for collecting data about the studied context and choosing the most suitable research design. The research team decided to base the research model on a longitudinal study together with supplementary follow-up interviews.

Secondly, an invitation to participate in the study was distributed to the workshop participants. Following the guidelines provided by Ployhart and Vandenberg [67], to those 43 developers who approved to participate in the study, we emailed educational material (see Appendix D) intended to minimize inter-observer (all researchers communicate the same knowledge) and inter-instrument variability (all participants receive the same information).

Since the definition of TD is crucial and sets the context for the entire study, we specifically focused on the educational material on guiding the respondents to understand the basic concepts of TD fully. Since there is no unified description of TD, we selected three different descriptions of TD, in order to provide the participants with information that they could relate to from their own perspective. Even if the selected different descriptions provide similar information, they also illustrate different aspects of TD, such as consequences, artifacts, costs, and life-cycle perspectives. First, the initial and commonly used definition of TD was introduced by Ward Cunningham's [186]: "Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite… The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on the debt," followed by Steve McConnell's also commonly used definition of TD [200]:"A design or construction approach that's expedient in the short term but creates a technical context in which the same work will cost more to do later than it would cost to do now (including increased cost over time)." Finally, a third description provided by the researchers of this study was used in order to provide a description of TD which illustrates that TD does not only have to include code- or design-related artifacts: "Technical debt is a non-optimal solution in code (or other artifacts related to software development) that gives a short-term benefit but cause an extra long-term cost during the software life-cycle."

14.3.2. Phase 2—Qualitative Data Collection

In the second phase, we conducted four rounds of interviews. This part of the study employed semi-structured interviews, including a mixture of open-ended and specific questions designed to elicit not only the information foreseen but also unexpected types of information [170]. The questions in the interviews were planned but not necessarily asked in the same order as they were listed. This interview technique allowed for the flexibility to explore interesting insights as they emerged. Each interview lasted between 30 and 45 minutes, and all interviews were digitally recorded and transcribed verbatim. All interviewees were asked for recording permission before starting, and they all agreed to be recorded and to be anonymously quoted for this paper.

We prepared a set of questions based on the antecedent factors of morale suggested by [198], and the interview protocol is available in Appendix B. To improve the reliability

المنارة للاستشارات

www.m

of the collected data, at least two authors participated in the interviews. In these interviews, when possible, we asked the interviewees to show us at least one specific item from their TD backlog and answer our questions by considering that item. For example, we asked all the interviewees from two of the companies to pick some of the existing issues with its SonarQube code-analysis tool, while developers from another company were asked to choose some of the non-allowed dependencies highlighted by their in-house tool. When selecting these specific cases, we tried to verify that these TD issues were affecting the interviewees' work in practice.

14.3.3. Phase 3a—Analysis and synthesis.

To analyze the qualitative data collected from interviews conducted in phase two, we used a thematic analysis approach [214]. Thematic analysis is a reliable data analysis method for capturing and reporting themes—important patterns of meaning related to a research phenomenon within qualitative data. Thematic analysis is especially suitable for studying the attitudes and behavior of people to explore a novel research phenomenon [215]. In this study, we conducted a deductive thematic analysis, in which the researcher codes data according to a pre-existing coding frame rooted in previous theories [214]. When analyzing the qualitative data, the guidelines provided by Braun and Clarke [214] were used to conduct the analysis in three phases using a thorough and rigorous approach.

In the first phase, we prepared a codebook based on two main sources that were initially identified. First, using the theoretical framework suggested by Hardy [198], a set of codes related to three dimensions of morale was generated.

Second, based on the results of Li, Avgeriou, and Liang [15], a set of codes related to TD occurrence (i.e., Causes of TD and Consequences of TD) and its management was generated. For instance, the codes TD Identification, TD communication, TD measurement, TD monitoring, TD prevention, TD prioritization, TD repayment, and TD representation-documentation, corresponding to TD management activities suggested by Li et al. [15], were capture how the interviewees manage TD. The full list of codes, second-order themes, and themes are shown in Appendix A. After preparing the codebook, the first author transcribed the recorded interviews, and the research team reviewed the transcriptions to familiarize themselves with the data and to get an overall idea of the collected data. The interviews with their transcriptions were added to a data analysis tool called NVivo.

In the second phase, the second author coded the interviews to identify data segments relevant to the research questions. Several of these initial codes were randomly picked and analyzed independently by the other authors, to triangulate the interpretation of the

المنارة للاستشارات

www.m

data and to minimize bias as much as possible. The coding procedure was reviewed by all the authors, and any conflicts were discussed jointly until an agreement was reached.

> "if you don't reduce the technical debt you will get development into a stall in some point; maybe we are actually walking now but we could run but the crawl will come if we don't pay-off the technical debt so we will get less and less feature in" – Interviewee 6

Fig. 3 Coded as "consequences of TD" and also as "Lack of progress," a code for low morale belonging to second-order theme "Progress" (see Appendix A)

We continued the data analysis process by assigning the coded extracts of data to all the relevant themes. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the code shown in Fig. 3 was assigned to two second-order themes, "Progress" and "Consequences of TD," which are shown in Appendix A. Thus, we could capture the relationship between themes to create our initial thematic map. Later in this phase, the research team reviewed the identified themes, their relations, and coded data assigned to each of the themes. Based on the feedback from reviewing the themes, the second author continued to refine the codes and themes and the thematic map. During this phase, we realized that the data from the interviews did not support some of the original themes shown in Appendix A, and therefore, the thematic map was updated accordingly.

Finally, we prepared a discussion of the results of our data analysis to explain the relations between the occurrence and management of TD and the developers' morale. At the end of this phase, we put forward a set of propositions about the influence of TD and its management on antecedents of morale.

14.3.4. Phase 3b—Quantitative Data Collection.

To complement our data and to clarify further the initial results from the interviews, we decided to conduct an online survey. In doing so, we designed a web survey that was hosted online by SurveyMonkey.com. The first draft of the survey was tested by the second author and one project manager to evaluate the understanding and the ordering of the questions and the usage of common terms and expressions [77]. During this evaluation, we also monitored the time that was needed to answer the questionnaire. The survey is shown in Appendix C.

The survey was accessible between December 1st, 2016, and January 31st, 2017, and a reminder was sent out after two weeks to all the invited participants. The survey was anonymous, and participation in the survey was voluntary. In designing the survey, we tried to form neutral questions and ordered and formulated them in a way that one question did not influence the response to the next question. The survey invitation was mailed directly to 34 developers in companies within our networks, all located in Scandinavia, with an extensive range of software development projects.

المنارة للاستشارات

www.r

The first part of the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their software. The second part of the survey included questions based on our theoretical propositions. As it can be seen from Table II, the participants were asked to rate a set of 5-point Likert Scale statements (very slightly or not at all, a little, moderately, quite a bit, extremely), to indicate their opinion about the impacts of TD and its management on antecedents of morale.

TABLE II - STATEMENTS RATED BY SURVEY RESPONDENTS

SID	Statements	The dimension of morale (second-order theme)
ST1	I have been criticized by others for taking TD.	Affective (Support and Communication)
ST2	I feel confident when I make a decision, which leads to TD.	Future/Goal (Vision for future)
ST3	I feel that the presence of TD hinders me from making progress.	Future/Goal (Progress)
ST4	I feel upset when others find out that I have taken TD.	Affective (Self-worth)
ST5	I feel that others appreciate it when I pay back some TD.	Affective (Support and Communication)
ST6	I feel satisfied when I pay back some TD.	Future/Goal (Progress)
ST7	I am encouraged by others to pay back TD.	Interpersonal (Influence of others)
ST8	I feel that paying back TD increases our team's morale.	All three

The statements aimed at capturing the opinions of survey respondents about our interpretations of the interview data. As such, in creating the statements, we tried to look at the codes which were assigned to our second-order themes and choose clear keywords with a minimum chance of misinterpretation. For example, as can be seen in Appendix A, *Criticism* is a code that is assigned to the second-order theme called *Support and Communication*, which belongs to the *Affective antecedents theme*. Therefore, in ST1, we asked the participants whether they have been criticized by others for taking TD.

المنارة للاستشارات

www.r

14.3.5. Phase 4—Analysis and synthesis.

In the fourth phase, the data collected from the previous phase (3b) were analyzed quantitatively, that is, by statistical analysis of the data collected from the survey answers. For descriptive purposes, data were summarized by mean, median, and standard deviation for continuous variables and numbers and percentages for categorical variables. We also used statistical methods such as Kendall's tau-b and Spearman's rank correlation coefficient to assess the strength and direction of the association between different variables.

14.3.6. Phase 5—Data Collection—Longitudinal study

The goal of this phase was to collect information on how much working time developers report as wastage due to experiencing TD.

A longitudinal study is a research method involving the collection of repeated observations of the same variables (in our case, the amount of wasted time) on more than one occasion [67] and over time [86]. The motivation for using this research method was mainly twofold:

a) We aimed to increase the precision of reporting experienced data (in our case, not based on single estimations or single perceptions of the wasted time). This was achieved by studying each respondent over several weeks using a repeated reporting design [86].

b) We aimed to survey respondents' changing reports over time: Longitudinal design has value for describing both temporal changes and their dependence on individual characteristics [86]. Further, the longitudinal research method increases the precision of measuring and reduces inter-individual variation.

This data collection phase included three steps (with three individual and unique sets of surveys), where the first step focused on respondents' background data, the second on the amount of time the respondents wasted due to experiencing TD, and the third on developer morale (due to TD). SurveyMonkey.com hosted all quantitative data collection online during the longitudinal study.

The first step was a start-up survey collecting descriptive statistics to summarize the characteristics of the respondents and their companies.

The second step in the longitudinal phase collected repeated reporting of the wasted time due to experiencing TD. This stage was designed to collect reported data from 43 software developers at 14 different survey occasions (i.e., twice a week for seven weeks) from October to November 2016. In total, 473 data points were collected, and each respondent reported their wasted time, on average, 11 out of 14 occasions. In this step, the respondents reported their data (wastage of time) to an online survey twice a week. To have equal spacing between the reporting occasions, as suggested by Morrison [167], for those respondents who did not answer within one day, a reminder was emailed.

During the entire period of this phase in the longitudinal study, the participants were asked to report their answer to the same survey question "**How much of the overall development time have you wasted due to technical debt (TD) since the last time you**

المنارة للاستشارات

www.r

took the survey?" Meaning the participant kept track of and calculated their own individual amount of wasted time.

In the surveys, the respondents reported the amount of wasted time using a value between 0-100 percent of their overall working time since they last took the survey. To address the potential problem with missing data from the respondents, if, for some reason, the respondents did not enter the data in one or more surveys, the respondents were asked to report their waste of time since the last time they took the survey. This means that if the respondent did not answer one or more surveys, the respondent would report the data from the last time the survey was taken. This means that reporting in the surveys cover the full period of sampling.

The **third step** of the longitudinal data collection phase was a follow-up survey to collect information indicating their opinion about the impacts of TD and its management on the antecedents of morale. This survey included questions based on our theoretical proposition presented in Fig. 1. In this part of the survey, participants were asked to rate a set of 5-point Likert Scale statements (very slightly or not at all, a little, moderately, quite a bit, extremely).

14.3.7. Phase 6—Analysis and Synthesis.

The data collected in the sixth phase were analyzed quantitatively, that is, by interpreting the numbers collected from the survey answers. All statistical analyses were performed with SPSS (version 22) and R version 3.3.2, using Tidyverse [216] version 1.1.1.

During this phase, we collected 473 data points (e.g., reporting of the amount of wasted time), where each developer reported on average 10.73 times out of 14 possible occasions (with a median of 12 and std. dev. of 3.91 times) with the average time interval between the reporting occasions of 3.1 days (with a median of 2.7 and std. dev. of 1.3 days).

In this phase, we further analyzed the association between the wasted working time and the antecedents of morale (section 2.2). The goal of this statistical analysis was to primarily indicate correlation, not causality, where the purpose was to understand if the quantity of wasted time, directly generated by TD, was associated with how the respondents perceived TD with respect to morale. For example, were respondents more inclined to consider TD affecting their morale if they wasted more time because of it? This was a way to analyze additional, quantitative, evidence to support (or not) our results. Such a step could be considered as increasing source triangulation of our study, improving its validity. Accordingly, for the test of normality (see Results section), we used a non-parametric method, called Spearman coefficient rank.

14.4. Results

The following subsections present the results for the research questions presented in Section 1. First, we present the results from the analysis of the qualitative data collection for each of the three identified dimensions of morale. Secondly, we present the results from the quantitative data collection.

المنارة للاستشارات

www.r

14.4.1. Qualitative data—Influence of TD on morale (RQ1, and RQ2)

In total, we conducted 15 face-to-face interviews with software professionals from five different companies (see Table III). Of the 15 interviewed respondents, we gained access to information about their working experience within the software development field. Most of the interviewees were relatively experienced, with a minimum of 5 working years, a maximum of 44 years and a median of 11 years.

Company A develops equipment for aerospace crews, and company B is an international telecom company. Both companies are located in Sweden. Company C is a medium-sized Finnish firm creating and providing a wide range of clinical data-assessment solutions to customers active in the healthcare sector. Company D is a German branch of an international enterprise providing software development and maintenance services to major aerospace customers such as the European Space Agency. Finally, company E is a Portuguese SME company providing a wide range of IT solutions to both public and private sectors in different areas such as telecommunications, gaming, finance, and so forth. The size of the companies cannot be presented due to the risk of violating their anonymity.

TABLE III - CHARACTERIZATION OF RESPONDENTS

ID	Role	Experience (years)	Company	Country	Domain
I1	Software Developer	NA*	A	Sweden	Aerospace
I2	Software Developer	28			
I3	Software Developer	44			
I4	Software Developer	7	B	Sweden	Telecom
I5	Software Developer	8			
I6	Software Developer	26			
I7	Testing Engineer	11	C	Finland	Healthcare
I8	Software Engineer	6	D	German	Aerospace

المنارة للاستشارات
www.r

ID	Role	Experience (years)	Company	Country	Domain
I9	Software Developer	21	E	Portugal	Healthcare
I10	Lead developer	10	E	Portugal	General IT
I11	Software Developer	9	E	Portugal	Telecom
I12	Project Manager	13	E	Portugal	Gaming
I13	Senior Architect	NA*	E	Portugal	Telecom
I14	Software Developer	11	E	Portugal	General IT
I15	Software Developer	5	E	Portugal	Gaming

* The interviewee preferred not to share the work experience

The results from the interviews suggest that both the presence and management of TD influence developers' morale to some extent. In particular, the occurrence of TD mainly has a negative influence on the affective and future/goal antecedents of morale. However, our results do not show any considerable impact of TD occurrence on the Affective and Interpersonal dimensions of morale. During the thematic analysis, we realized that the interview data do not provide enough evidence to saturate some of the original second-order themes and main themes related to TD occurrence. These themes and their relationships are shown in gray in our final thematic map in Fig. 4.

المنارة للاستشارات

www.r

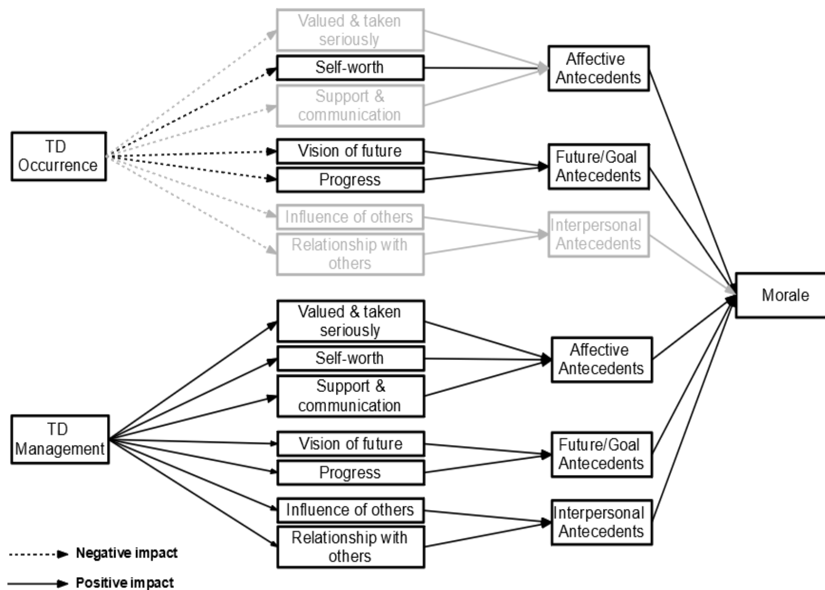


Fig. 4. Visualization of the final thematic map

On the other hand, the interview data indicate that the management of TD has a strong positive influence on morale. In particular, the interview data show that TD management has a clear positive influence on all the second-order themes and consequently forming the three dimensions of morale. In the following sections, we discuss these findings in more detail.

14.4.1.1. Affective antecedents (RQ1 and RQ2)

To explore the influence of TD and its management on the affective dimension of morale, we assigned codes into three second-order themes (i.e., valued and taken seriously, self-worth, and support and communication). However, from the interviewees' data, we could only identify a clear relationship between the occurrence of TD and self-worth and a weak link to support and communication. In other words, it seems that the occurrence of TD only has a negative influence on developers' self-worth, but they do not have any strong affective reaction to it.

Most of the interviewees mentioned that they do not think that their colleagues or management team have ever criticized them for introducing TD. However, some of them mentioned that since being criticized could be unpleasant, an individual developer might hesitate to criticize their teammates for taking on TD. Thus, it is very important to communicate others' mistakes appropriately. Interviewee 3 raised this issue as shown in the following excerpt: "*You would have to go and ask them why it is all red in your [SonarQube] panels when mine is green, but that is a question that is a bit difficult to ask if people get upset if you hint that they have bad quality*."

المنارة للاستشارات

www.r

Even though most of the interviewees think that they have not been subject to criticism and they should not blame others for taking on TD, several interviewees mentioned that they do not feel good about making a decision, which obviously leads to the occurrence of TD. The experience of realizing that they have introduced TD is explained by several interviewees as a negative feeling by using a wide range of negative terms such as a sense of insecurity and failure or being afraid and upset. For instance, interviewee 2 mentioned this issue this way: "*It is some kind of a bad feeling because it nags me. […] if I have missed something that I feel that I should have known*." Several interviewees mentioned that the reason for such feelings is that they tend to perform a good job. On the other hand, some of the interviewees mentioned that working with software artifacts, especially legacy code, which contains a large amount of TD is scary and frustrating. It must be noted that three interviewees mentioned that such negative feelings might depend on the level of work experience. Based on the above discussion, we put forward the following proposition1:

PTD1: The occurrence of TD may have a negative influence on the affective antecedents of morale.

On the other hand, it seems that the management of TD influences all the second-order themes of affective dimension. First, the majority of the interviewees mentioned that their efforts for managing TD are valued and appreciated by their colleagues and management team. For example, interviewee 4 mentioned that "*if you deal with technical debt that gives a measurable surface impact for improvement […] those things are highly appreciated".*

Additionally, the majority of our interviewees think that others trust their decisions concerning TD management, while half of them mentioned that they have a reasonable amount of autonomy to perform refactoring and repaying TD. However, it seems from the interviewees' perspective that the companies do not reward TD management sufficiently. One developer even mentioned that in their company, slight praise or appreciation is missing, while another said that if developers cannot show its value and results, others do not appreciate repaying TD.

On the other hand, the interview data indicate that managing TD is associated with a sense of self-worth among software developers. The majority of the interviewees think that the identification and communication of TD enable them to improve their skills by understanding their mistakes and learning new aspects of software engineering. Interviewee 1, for instance, mentioned that "*over the years [it had] been teaching me lots of stuff and made me a better developer in a shorter time than would have been possible without it. I think it has been a huge change*." Also, it seems that managing TD leads to a sense of achievement and success among developers. Several interviewees mentioned that performing refactoring and repaying TD is an interesting activity, which motivates them as well as enables them to increase the quality of software artifacts constantly and feel confident about their products.

---

1 In acronyms used for propositions, PTD stands for Proposition Technical Debt and PTDM stands for Proposition Technical Debt Management.

المنارة للاستشارات

www.m

However, several interviewees mentioned that managing TD is not an easy and straightforward task, and it may be considered pointless. Managing TD of legacy code is especially challenging since developers often need to perform a long chain of unexpected changes in different parts of the code until they get to fix the initial issue. Also, it is hard to show the real value of such improvements and convince managers why it is necessary to perform them. Several interviewees considered repaying TD to be unnecessary or even dangerous in some cases since it might cause failure in a working system. Therefore, some developers might prefer not to touch the code since they consider repaying TD to be pointless.

Finally, the majority of our interviewees mentioned the good support and communication within their company for TD management. It seems that they consider TD as an inevitable phenomenon, which must be identified, documented, and repaid as time and resources allow. In other words, no matter who introduces TD, they consider TD management a task that everyone can benefit from and necessary for improving the quality of the software artifacts.

Overall, based on the observations discussed in this section, we propose that TD has a negative influence on affective antecedents of morale, while TD management has a positive influence on affective antecedents of morale. Overall, based on these observations, we propose that:

PTDM1: TD management has a positive influence on the affective antecedents of morale.

14.4.1.2. Future/Goal antecedents (RQ1 and RQ2)

To explore the influence of TD and its management on the future/goal dimension of morale, we assigned codes into two second-order themes (i.e., the vision of future and progress). Based on the interview data, the influence of TD on the future/goal antecedents of morale seems to be strong. The majority of the interviewees mentioned that the presence of TD has a negative influence on their progress and future activities. In particular, they mentioned that the presence of TD hinders them from progressing in their tasks. At the same time, it makes the future unclear since developers may face unpredicted difficulties.

As mentioned by several interviewees, large amounts of TD make the maintenance difficult and costly since understanding and working with the software becomes problematic, and there might be unpredictable issues. Also, if TD is not paid back, the development speed can decrease, and adding new features in the future becomes very challenging and even impossible. In the following excerpt, interviewee 6 talks about such an issue: "*If you don't reduce the technical debt, you will get development into a stall in some point; maybe we are walking now, but we could run, but the crawl will come if we don't pay off the technical debt*."

On the other hand, some of the interviewees mentioned that the presence of TD brings developers' confidence down. This could either be because of their awareness of a large

المنارة للاستشارات
www.r

volume of TD in their artifacts or because it hinders developers from performing their tasks. Several interviewees mentioned that introducing TD could make them feel that they lack the necessary skills and lower their confidence. As a result, they start to be more conscious about their decisions, and therefore, development speed slows, or developers will be reluctant to perform refactoring to improve their code.

Several interviewees mentioned the influence of past decisions leading to the occurrence of TD on their progress. Even though they did not blame previous developers for taking on TD, some of the interviewees mentioned that they have to deal with TD, which is the result of poor decisions made by previous developers and which could have been avoided. For instance, interviewee 8 mentioned that "*in the past, either we or that company before us did not really pay attention and violated the guideline, and because of that, we have to pay back [TD] now.*"

Also, several interviewees mentioned that when different modules are developed by different developers or teams, the existence of TD in one module can influence the progress of other developers as well. As a result, working with an external module, which is not well maintained, is often frustrating and time-consuming. Based on the above discussion, we put forward the following proposition:

PTD2: The occurrence of TD has a negative influence on the future/goal antecedents of morale.

On the other hand, the majority of the interviewees mentioned the importance of TD management, which, from their perspective, makes the future better than the present. These interviewees mentioned that they have a clear vision of how to manage TD properly within their company. In doing so, they use different techniques and tools to identify, document, communicate, prevent, and repay TD. Also, it seems that proper TD management is associated with a sense of satisfaction. For instance, interviewee 6 mentioned that "*I think that is a nice feeling, to be able to pay your debt [...] I would say it is a positive feeling; it is a motivator.*"

All the interviewees consider repaying TD to be necessary, but for some items, it is pointless. For instance, several interviewees mentioned that repaying architectural TD and testing TD is very important, while some of them mentioned that it is pointless to fix issues which are considered to be "cosmetic" (e.g., documentation TD) or the items which are mistakenly highlighted by tools (i.e., false positives), or TD items located in those parts of the legacy code that rarely change. However, during the data analysis process, we identified controversial opinions about such "cosmetic" things and "false positives." For instance, one software developer suggested that following simple rules such as naming conventions is very important, while another one suggested that it does not make sense to go back and fix naming issues. In another case, one interviewee suggested that writing good comments facilitates future maintenance of software, while another one mentioned that repaying documentation TD is not worth spending the time and effort.

From the interview data, it seems that proper TD management leads to a sense of progress among software developers. The majority of the interviewees consider TD identification and communication a contribution to their teams' goals, and by repaying TD, they feel

المنارة للاستشارات
www.m

successful in progressing toward those goals. Such a sense of progress is a positive feeling, which enables developers to perform their tasks easily and smoothly.

Therefore, based on the above-mentioned observations, we propose that the occurrence of TD has a negative influence on the future/goal antecedents of morale, while TD management has a positive influence on the future/goal antecedents of morale. Therefore, based on these observations, we propose that:

PTDM2: TD management has a positive influence on the future/goal antecedents of morale.

14.4.1.3. Interpersonal antecedents (RQ1 and RQ2)

To explore the influence of TD and its management on the interpersonal dimension of morale, we assigned codes into two second-order themes (i.e., the influence of others and relationships with others).

Regarding the relationship between the influence of others and the occurrence of TD, the majority of our interviewees do not blame their colleagues or previous developers for introducing TD even though they acknowledge that previous sub-optimal decisions which have led to the occurrence of TD could be avoided in the past. The following excerpt shows the opinion of interviewee 5 in this regard: "*It maybe is stressful or frustrating to work with this legacy code; if [it] had been written in a better way it would have been much faster and easier to implement this feature*."

Regarding the relationship with others, it seems that from the interviewees' perspective, there is very good cohesion and understanding within their teams about the reasons behind the occurrence of TD. Most of the interviewees think that a combination of factors leads to the occurrence of TD, and therefore, they do not feel that their teammates or previous developers dragged them down by introducing TD. This seems to be the case, especially in those teams where team members have a closer relationship, as interviewee 10 mentions in the following excerpt: "*We should not point a finger or place a picture on the wall, 'this guy broke some rules.' And I also believe that a significant part of company's success is directly motivated by [and] directly related to our engagement as colleagues in our team because we say we are all friends here; we know each other; three of us know each other since college days, and we are friends also outside.*"

Our interview data suggest that the decision to take on TD is often made based on a consensus among team members, especially in smaller teams. For instance, one interviewee stated that "*if we have a short time frame sometimes [and need to] make a decision, do we go to technical debt and let these things go like this so we can keep the deadline or not? We discuss that usually. In the end, if the team has a consensus, there are no problems, but if there is no consensus in the team, the lead developer—the senior one—makes a deciding vote*."

Since we could not discover strong evidence from our data indicating that TD has a negative influence on interpersonal antecedents of morale, we put forward the following proposition:

المنارة للاستشارات
www.r

PTD3: The occurrence of TD has no negative influence on the interpersonal antecedents of morale.

On the other hand, regarding the influence of TD management on the interpersonal dimension of morale, it seems that the majority of our interviewees consider TD management as teamwork by which they contribute to a set of common goals within their teams. They especially consider conducting reviews as a positive contribution by which they can identify TD and help each other in improving their skills and, ultimately, the quality of software. In the following excerpt, interviewee 10 refers to this matter: "*Well, I appreciate that review. I always try to learn, and learning from my mistakes is also something that I do when I keep doing some self-evaluation of my team, and that's one way my evaluation refines. Yes, I appreciate that someone evaluates my work generally, and the code is just one of the parts.*"

On the other hand, regarding the relationship with others, most of our interviewees think that the current team is responsible for managing TD and improving the quality of software constantly, even though a few of them mentioned that it is not pleasant to improve the legacy code. This may be due to the challenges of repaying the TD of the legacy code, as discussed earlier. Therefore, they think that identification and communication of TD is a service, which is done to help their teammates in improving the quality of software artifacts. In the following excerpt, interviewee 2 points to this: "*I will [report it] so they have to fix it or to have a more thorough discussion about it, why it happened and why you did that if it can't be fixed. But mostly I feel that it is a service; we do each other that kind of service that together we can make better code*". Based on the above discussion, we propose that:

PTDM3: TD management has a positive influence on the interpersonal antecedents of morale.

To summarize, the interview data provide evidence that TD can have a negative influence on affective and future/goal antecedents of morale but no influence on interpersonal antecedents of morale. On the other hand, TD management has a positive influence on all three dimensions of morale, especially on the future/goal dimension.

14.4.2. Quantitative data—Influence of TD on morale (RQ1, and RQ2)

In total, we received 34 valid responses to the survey from developers across seven software companies. Please note that two of these respondents work at Company A, five work at Company B, and the rest (i.e., 27 respondents) work in other companies from which we did not collect any interview data.

Initially, the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their software. This data is compiled and presented in Table IV. As can be seen from this table, all the survey respondents have completed a university degree,

المنارة للاستشارات
www.m

while the majority of them have more than 10 years of work experience in the software industry.

TABLE IV - CHARACTERISTICS OF THE SAMPLE SURVEY

Individual		Company	
Experience		**Software system type(s)**	
< 2 years	8.8%	Embedded Sys.	76.5%
2 - 5 year	17.6%	Real-time Sys.	29.4%
5 - 10 year	17.6%	Data management Sys.	5.9%
> 10 years	55.9%	System Integration	2.9%
Education		Modeling and/or simul.	2.9%
Master's degree	76.5%	Data analysis sys.	14.7%
Bachelor's degree	20.6%		
Ph.D. degree	2.9%		
Gender			
Male	82.3%		
Female	17.7%		

As illustrated in Table IV, all the respondents were relatively experienced as software developers: 56% had more than 10 years of experience, and only 7% had fewer than 2 years of experience. All respondents have a university-level education, where 82% have a master's degree.

Further, the results from the survey assessing the impact of TD on the different dimensions of morale show that the occurrence of TD negatively influences the future/goal and affective antecedents of morale but not its interpersonal antecedents, while the management of TD positively affects all the three dimensions of morale.

When studying the correlation between the amount of time that is wasted due to TD and the developer morale, our result shows that the more time that is wasted, the more the respondents feel that the progress is hindered by TD. In the following sections, we discuss these findings in more detail.

14.4.2.1. Survey result (RQ1 and RQ2)

As described in section 3.1, we complemented our findings from the interviews by conducting a survey in which 33 software professionals rated a set of 5-point Likert Scale statements (see Table V). Utilizing our theoretical framework and findings from the interviews, we prepared these statements in a way that they complement our findings from the interviews. As illustrated in Table V, four of these statements are related to the negative effects of TD (ST1, ST2, ST3, ST4), and three of them are related to the positive

المنارة للاستشارات

www.r

effects of TD management (ST5, ST6, ST7) on antecedents of morale. Finally, ST8 directly refers to the positive influence of TD management on morale. This enables us to verify whether respondents' ratings about the influence of TD management on antecedents of morale are in line with their perception of the influence of TD management on morale itself. Fig. 5 shows a summary of the ratings provided by the respondents.

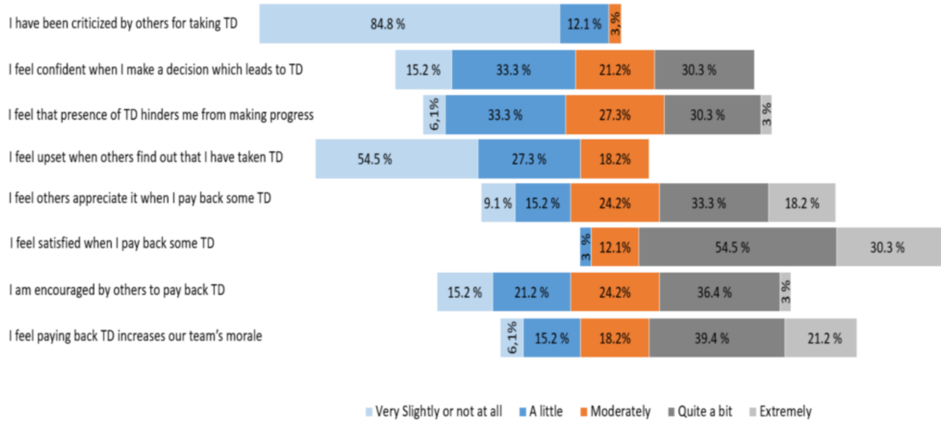


Fig. 5 Summary of the responses to the online survey.

TABLE V - MEDIAN AND IQR CALCULATED FOR SURVEY RESPONSES

SID	Statement	Median	IQR	Supports the interviews
ST1	I have been criticized by others for taking TD.	1	0	Yes
ST2	I feel confident when I make a decision, which leads to TD.	3	2	Yes
ST3	I feel that the presence of TD hinders me from making progress.	3	2	Yes
ST4	I feel upset when others find out that I have taken TD.	1	1	Yes
ST5	I feel that others appreciate it when I pay back some TD.	4	2	Yes
ST6	I feel satisfied when I pay back some TD.	4	1	Yes
ST7	I am encouraged by others to pay back TD.	3	2	Yes
ST8	I feel that paying back TD increases our team's morale.	4	1	Yes

المنارة للاستشارات

www.r

Table V shows the central tendency (i.e., median) and frequency (i.e., interquartile range) of the respondents' ratings. It must be noted that a greater median indicates that the respondents consider the influence mentioned in a statement to be more significant, while a smaller median indicates such an influence to be less significant. On the other hand, a lower interquartile range (IQR) value indicates higher levels of consensus among respondents, while a higher IQR value implies lower consensus among respondents.

Regarding the influence of TD on the affective antecedents of morale, ST1 received a very low rating (median = 1) with a very high level of consensus among respondents (IQR = 0). This suggests that the respondents think that they rarely have been subject to criticism for introducing TD. Moreover, ST4 received a very low rating with a high level of consensus among respondents (median = 1; IQR = 1). These observations suggest that the respondents consider the affective consequences of TD to be very insignificant, which is in line with our interview data. Therefore, we can conclude that the occurrence of TD has a negative influence on developers' morale since it reduces their confidence.

Regarding the influence of TD on the Future/Goal dimension of morale, ST2 received a moderate rating (median = 3), which suggests that the majority of respondents, with a medium level of consensus (IQR = 2), do not feel very confident when introducing TD. Moreover, the ratings for ST3 (median = 3; IQR = 2) show that the survey respondents, with a medium level of consensus, consider TD to hinder them moderately from making progress. These results support our findings from interviews, which indicate that the occurrence of TD has a negative influence on Future/Goal antecedents of morale. In other words, we can say that the occurrence of TD primarily has a negative influence on developers' morale because it hinders their progress.

On the other hand, concerning the impacts of TD management on antecedents of morale, ST5 received a high rating (median = 4) with a medium level of consensus among respondents (IQR = 2). This observation supports our interview data and suggests that the survey respondents generally think that their efforts for managing TD are appreciated by their colleagues. Therefore, it can be said that TD management has a positive influence on developers' morale since it is associated with a sense of appreciation. In addition, ST6 received a high rating from respondents with a high level of consensus (median = 4; IQR = 1), which indicates that the respondents strongly feel satisfied with repaying TD. This supports our interview data, which suggests managing TD has a positive influence on future/goal antecedents of morale since it is associated with a sense of progress. In other words, TD management has a positive influence on developers' morale since it is associated with a sense of progress.

As can be seen from Table V, ST7 received a moderate rating (median = 3) with a medium level of consensus (IQR = 2). This shows that the respondents think that they are moderately encouraged by others to repay TD, which indicates the positive influence of TD management on interpersonal antecedents of morale. This observation is in line with our findings from the interviews. In particular, we can conclude that TD management has a positive influence on developers' morale since it is encouraged by others. Finally, ST8 was commonly rated as significant (median = 4; IQR = 1), which alongside interview

المنارة للاستشارات

www.r

data, suggests that, in general, management of TD has a positive influence on developers' morale. Based on these results, we are confident that our propositions about the influence of TD management on different dimensions of morale are plausible, and therefore we put forward another proposition as:

PTDM4: TD management is perceived to have a positive influence on the team's morale.

To summarize, the occurrence of TD negatively influences developers' confidence and their sense of progress and consequently lowers their morale. On the other hand, management of TD is associated with a sense of progress and a sense of appreciation and support from others, which raises developers' morale. To further investigate these findings, we performed a set of statistical tests to identify any potential correlations between our propositions. This would show if and what kind of relationships there might be among the different dimensions of morale affected by TD occurrence or management as well as to verify any potential influence on our results of two additional factors (i.e., developers' level of work experience and the amount of waste that occurs due to TD) identified from the interview data.

Before analyzing the association between wasted time and the antecedents of morale, we first tested the variables for normality using the Shapiro-Wilk test of normality (see Table VI). None of the variables are normally distributed, as the null hypothesis (i.e., the data points of the variable are extracted from a normally distributed population) is rejected with very low p-values. This means that we cannot use Pearson, but we need to use non-parametric methods. Therefore, we use the Spearman's rank correlation coefficient.

The matrix of correlation in Fig. 6 shows only those associations that are significant (p-value < 0.1). The blank cells represent no correlations.

المنارة للاستشارات

www.r

TABLE VI - TEST OF NORMALITY FOR CORRELATED VARIABLES

Variable	Statements	Shapiro-Wilk test	p-value
Waste	Wasted time because of Technical Debt	0.80817	3.701e-05
ST1	I have been criticized by others for taking TD.	0.86125	0.0005045
ST2	I feel confident when I make a decision, which leads to TD.	0.74087	2.221e-06
ST3	I feel that the presence of TD hinders me from making progress.	0.89586	0.003573
ST4	I feel upset when others find out that I have taken TD.	0.48118	7.893e-10
ST5	I feel that others appreciate it when I pay back some TD.	0.88216	0.001602
ST6	I feel satisfied when I pay back some TD.	0.82243	7.176e-05
ST7	I am encouraged by others to pay back TD.	0.90563	0.00648
ST8	I feel that paying back TD increases our team's morale.	0.88989	0.002506

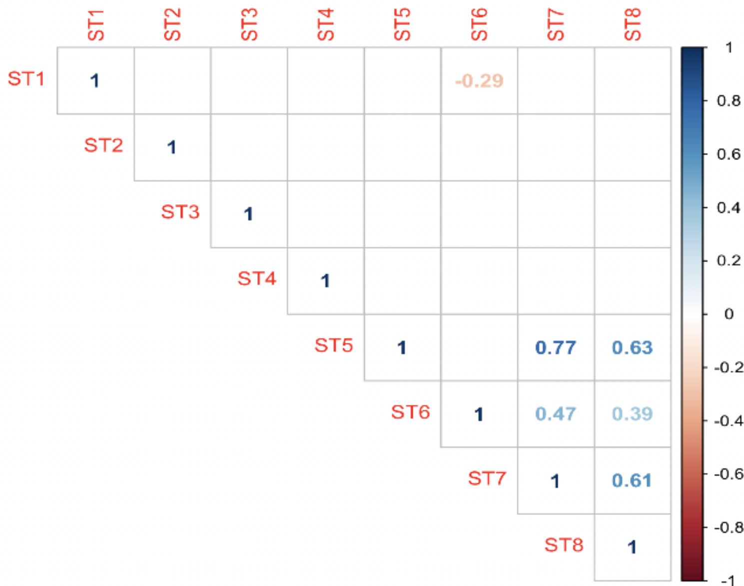


Fig. 6. Associations between the different statements

المنارة للاستشارات

www.r

We can see how most of the statements related to the occurrence of TD and morale were not statistically correlated among themselves or with the statements related to TD management. On the other hand, we can see how most of the statements related to the management of TD and morale are correlated among themselves. These results are further discussed in the following paragraphs.

- *The negative correlation between criticism (lack of support and communication) and sense of progress (ST1-ST6):* This significant negative correlation (-0.29, p-value < 0.1) indicates that the less criticism the practitioners receive because of taking TD, the more satisfaction they get from their progress in paying it back. On the contrary, this negative correlation may also be interpreted as though developers feel less satisfaction in paying TD back, but they might also perceive more criticism due to taking TD.
- *Correlation between a sense of appreciation and sense of support (ST5 - ST7):* This strong and significant (0.77, p-value < 0.05) correlation indicates that practitioners who feel encouraged to repay TD also feel that their efforts towards TD repayment are recognized by their colleagues. This could potentially mean that showing appreciation and recognition is a good method for encouraging developers to repay TD.
- *Correlation between a sense of progress and sense of support (ST6-ST7):* This medium correlation (0.45, p-value < 0.05) can be interpreted as the more encouraging support the respondents receive to repay TD, the more satisfied they are with their progress. The inverse implication could potentially be that if someone feels satisfied by paying TD back, they also feel encouraged by others to do so. On the other hand, since satisfaction could also be considered as an affective state which is associated with self-worth (i.e., second-order theme in affective antecedents), this correlation might be interpreted as the more encouraging support the respondents receive to repay TD, the better they feel about their achievements.
- *Correlation of sense of appreciation, sense of progress, and sense of support with the perceived team's morale (ST5, ST6, ST7, and ST8):* We discuss these correlations altogether because ST5, ST6, and ST7 are already correlated and have been discussed. From these correlations, we could infer that those practitioners who were encouraged to pay TD back, who were satisfied in doing so (although the correlation is less strong here) and felt like others would appreciate it, also think that paying back would increase team morale. One explanation might be that potentially all these factors contribute to better team morale or having better team morale makes others feel like they are encouraged, satisfied, and supported in repaying TD. In the second case, it could potentially be explained as "*we are happy in the team, so we feel like we are better at dealing with TD*," which might also be a possible explanation, although less supported by evidence from interviews.

المنارة للاستشارات

www.r

As discussed earlier, some of the interviewees mentioned that developers' perception of the consequences of TD could depend on their level of work experience. To determine such relationships, a set of *Kendall's tau-b* correlation tests were conducted. The results of these tests indicate that the only considerable correlation is between respondents' work experience and their perception about being criticized for introducing TD (i.e., ST1), which was positive and statistically significant at the 0.05 level ($\tau b = 0.338$, $p = 0.039$). This rejects the idea that less-experienced developers are more concerned about being subject to criticism for introducing TD and supports our previous findings indicating that, in general, developers rarely criticize each other for the occurrence of TD. Another interpretation could potentially be that the more experienced developers perceive a higher level of criticism in case others highlight their mistakes or their actions in taking TD.

14.4.2.2. Correlation between the wasted time (due to TD) with the TD occurrence and the TD management dimensions of morale (RQ3)

As discussed earlier, almost all our interviewees and respondents to the 5-point Likert Scale survey mentioned that the occurrence of TD hinders their progress and their future activities. Several interviews discussed the amount of development time and resources that are wasted because of the TD accumulated in their products. Therefore, we decided to investigate any potential correlation between the amount of waste that occurs due to TD and developers' morale.

First, we calculated the average amount of perceived wasted time reported by each participant during the second step of the longitudinal data collection (i.e., phase 5). Following this, we used Spearman ranking to study the correlation between the average amount of wasted time reported by each respondent, and their rating for the 5-point Likert scale statements. The association might tell us whether participants who reported a higher average waste of time would also rate higher the statements ST1-ST8 (the antecedents of morale related to the occurrence and management of TD). In other words, we want to understand whether more waste (the negative effect of TD) correlates with lower or higher morale (with respect to different dimensions)

TABLE VII - CORRELATION BETWEEN THE WASTE OF TIME AND THE STATEMENTS

% Waste correlation with:	**ST1**	**ST2**	**ST3**	**ST4**	**ST5**	**ST6**	**ST7**	**ST8**
Spearman (approx.)	-0.109	-0.036	0.453	0.061	-0.076	0.087	-0.202	- 0.070
p-value (approx.)	0.536	0.837	0.007	0.728	0.666	0.624	0.251	0.690

المنارة للاستشارات

www.r

As we can see from Table VII, the only significant correlation is between the average % of the wasted time and ST3 (“I feel that the presence of TD hinders me from making progress”). This indicates that the more time is wasted, the more the respondents feel that the progress is hindered by TD. This interpretation is supported by our observations from the interviews. However, the observed correlation can also be interpreted differently: The respondents with a lower level of progress, or possibly morale, might tend to estimate the amount of waste to be higher.

The lack of associations between the waste due to TD and the morale variables may not exclude the existence of relationships. But in this study, we do not find evidence indicating that the waste of time due to TD correlates with other morale variables than ST3.

14.5. Discussion

In this study, we aim to answer and discuss the earlier stated research questions. In response to RQ1, regarding the influence of TD on morale, it can be said that the occurrence of TD may reduce developers’ morale mainly because the developers perceive it to hinder their progress and makes it challenging to perform their tasks. In response to RQ2, it can be said that proper management of TD appears to increase developers’ morale since it is associated with positive personal and interpersonal feedback and enables developers to perform their tasks better and to improve software quality in the future. In RQ3, we assess how the amount of wasted time correlates with developer morale, and the results indicate a significant correlation between the productivity (e.g., the perceived wasted time) and the developers’ feeling that their progress is hindered by TD.

Although in recent years, considerable interest has been shown in researching TD, there is a lack of studies exploring this phenomenon from individuals’ perspective and particularly concentrating on explaining the social and psychological aspects of TD. By reviewing previous literature on TD, we could find only a limited number of previous studies mentioning that TD may influence developers’ emotions [10], [11], and morale [12],[7],[13],[201]. Therefore, we decided to conduct a field study to explore the potential impacts of TD and its management on developers’ morale.

Our findings show that the occurrence of TD is perceived to have a negative influence, mainly on the future/goal and affective antecedents of morale and not on its interpersonal antecedents. In other words, our results indicate that TD reduces developers’ morale since it hinders them from performing their development tasks, making progress, and achieving their goals. This is in line with previous studies by [11] that suggest the presence of TD slows development and reduces developers’ productivity [7]. Also, the lack of development resources is reported to hinder developers’ progress and seam to consequently lower their morale [184],[200]. These results further support our findings considering that resource constraints are one of the main reasons for incurring TD [217].

Our results also indicate that the occurrence of TD has a negative influence on affective antecedents of morale and, in particular, developers’ self-worth. In an exploratory field study and based on interviews with 35 software developers, Lim et al. [8] suggest that

المنارة للاستشارات

www.n

software developers have a negative attitude toward TD since they tend to create "perfect software." The results of another case study conducted by [10] at a Finnish software company show that software developers do not feel good about taking TD since they have to deal with it in the future. Finally, Peters [201] suggests that the presence of TD has a negative influence on developers' motivation. Even though none of the previous studies directly refers to the relationship between TD and developers' self-worth, this observation may be explained in terms of developers' tendency to produce high-quality software [8]. In particular, since the occurrence of TD could hinder developers from achieving this goal, it may have a negative influence on developers' self-worth.

Additionally, our results suggest that TD management has a positive influence on the antecedents of morale, and consequently, it can be said that proper management of TD might increase the developers' morale. McConnell [200] makes a similar argument in a blog post by suggesting that repaying TD "can be motivational and good for team morale." Finally, in their empirical study and based on the results of two surveys answered by 37 software professionals, Spínola et al. [12] suggest that repaying TD may have a positive influence on morale; however, future research is needed to clarify this relationship. Damian and Chisan [181] also suggest that continuous software improvements, in general, have a positive influence on developers' pride and morale.

Finally, in this study, we examine the relationship between TD and waste. Our result indicates that the more time that is wasted due to experiencing TD, the more the respondents feel that the progress is hindered by TD. Even though this interpretation is supported by the interview data, another interpretation is that the respondents with lower levels of progress tend to estimate the amount of waste to be higher. This may be explained in terms of confirmation bias whereby individuals tend to report information which confirms their viewpoints or beliefs. In other words, since developers believe that TD hinders their progress, it appears that they tend to estimate the amount of waste to be higher than its actual amount.

On the other hand, our results indicate a lack of correlations between the perceived waste due to TD and the interpersonal and affective dimensions of morale plausible (ST1, ST4, ST5, ST7, and ST8). In fact, we would expect such associations to show up if the TD was explicitly accrued by one member of the team (or at least an employee somewhat close to the team), while the interest would be paid by another. However, due to potential high turnover and the presence of external consultancy or heavy outsourcing, it is likely that developers find themselves dealing with the waste generated by code written by someone who is unknown to them. This could potentially explain why TD waste does not seem to be associated with interpersonal or affective factors.

On the other hand, it is still somewhat interesting to notice that our results do not indicate an association between the TD waste and ST2, ST6 (future goal dimension). In fact, we might have expected a relationship between the amount of TD waste and the confidence in taking on TD (ST2): the more waste is paid, the less one might feel confident in taking TD. However, this can potentially be explained by the fact that accruing TD alone does not generate waste: the waste could be caused by a lack of dealing with such TD after the accrual (lack of refactoring). As for ST6, one might expect the satisfaction of paying back

المنارة للاستشارات

www.m

TD to be associated with higher waste or lower waste (avoiding a high amount of waste by repaying the debt could generate more satisfaction). However, the two variables are not directly correlated, as one might pay back TD, while at the same time suffering from high waste from other TD items, which cannot be refactored by that specific person.

In general, while for ST3, there is a correlation between the presence of TD waste and lack of progress, the other variables cannot directly be linked to any form of generic TD waste, but only in specific circumstances. To further understand such associations, we would need to conduct specific studies where variables related to the affective state of participants are controlled or at least better known.

Based on our findings, it seems that developers' perception of the amount of time wasted on managing TD could trigger a vicious circle where the presence of TD leads to higher perceived waste, which continuously lowers developers' morale and productivity. However, it is important to acknowledge that the correlational analysis used when answering the research questions do not infer any causal relationships between the different studied variables and thus does not suggest any casualties.

This is in line with Tom et al. [7], who argue that TD lowers developers' morale, which in turn increases the amount of technical debt. This phenomenon can be explained by Hardy's [198] argument that "the consequences of morale feedback into the antecedents. This feedback loop seems to act as a form of confirmation bias whereby information which confirms the prevailing morale state is selected and that which refutes it rejected." Therefore, it can be said that low morale appears to be associated with a perceived lack of achievement and negative self-assessment [218],[213]. Thus, in the context of our study, when developers perceive TD to hinder their progress, their morale appears to declines and this, in turn, appears to cause them to assess their accomplishments lower and their waste higher. In other words, developers with low morale appear to have an over-exaggerated perception of the extent to which TD hinders their progress and the amount of time wasted on servicing TD. This, in turn, may lead to negative self-assessment (i.e., the developer experiences more waste), which lowers their morale even more, which consequently could worsen their perception of the time wasted due to experiencing TD (i.e., negative self-assessment).

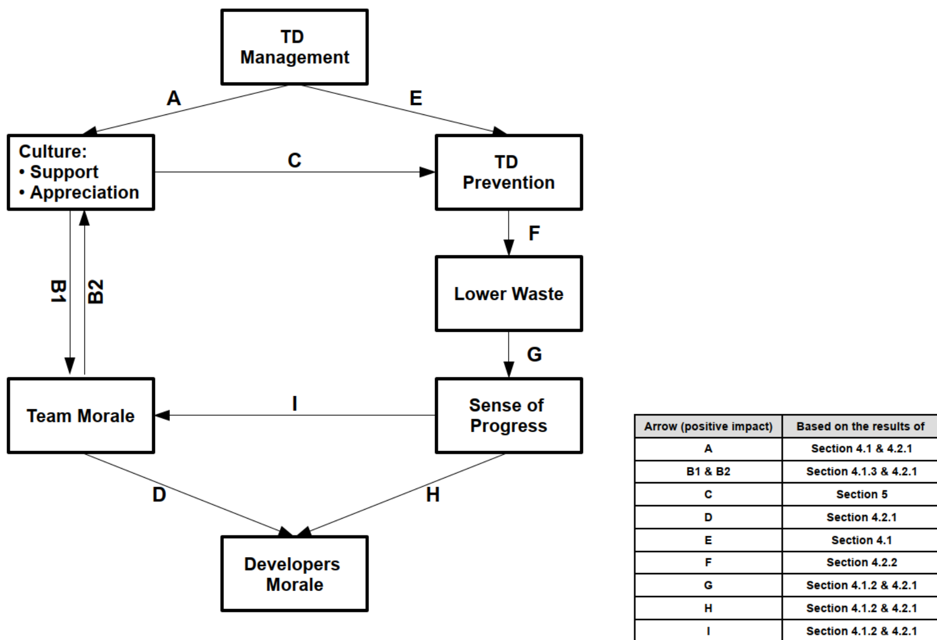
14.5.1. Implications

Our study has several novel contributions for both software engineering research and practice. First, our study is the first that specifically concentrates on investigating the influence of TD on developers' morale and its relation to developer productivity. In doing so, we explored the impacts of TD and its management on the antecedents of morale. As a result, we were able to show that the occurrence of TD can reduce developers' morale, while TD management increases their morale. These findings are very encouraging since they clarify the relations between different dimensions of morale and TD. In particular, by emphasizing the importance of affective and future/goal dimensions of morale, this study suggests that, while making trade-offs leading to the occurrence of TD, software firms must better consider the short- and long-term consequences of TD on developers' behavior and productivity as well as on software development and maintenance costs and

المنارة للاستشارات
www.r

success. This becomes even more important considering the interviewees' controversial viewpoints about the necessity of repaying different types of TD items. For instance, some of the interviewees considered fixing "cosmetic" issues (e.g., documentation TD) to be pointless, while repaying architectural TD and testing TD to be very important since long-term costs and complexities associated with them can surpass the short-term benefits gained by taking on such TD. On the other hand, the results of this study suggest that software firms need to consider TD management more seriously, by investing more resources and promoting a high-quality culture in which developers are encouraged and rewarded for identifying and repaying TD. Showing appreciation and recognition is a good method for encouraging developers to repay TD since it seems to create a sense of satisfaction for developers. In particular, we argue that if developers are supported and encouraged to repay TD, they feel more satisfied with their achievements. In general, we could infer that a company culture that is encouraging TD repayment also increases team morale.

Our suggestion is supported by the results of an empirical study showing that a lack of remuneration and reward schemes reduces developers' morale and productivity [183]. This need becomes more obvious considering previous research suggesting that the level of employees' morale is correlated with their productivity [181], [182], [183], and project success [184].



Arrow (positive impact)	Based on the results of
A	Section 4.1 & 4.2.1
B1 & B2	Section 4.1.3 & 4.2.1
C	Section 5
D	Section 4.2.1
E	Section 4.1
F	Section 4.2.2
G	Section 4.1.2 & 4.2.1
H	Section 4.1.2 & 4.2.1
I	Section 4.1.2 & 4.2.1

Fig. 7. Twofold model of the relationship between TD and morale

Based on the results of this study, we propose a twofold model for the description of the relationship between TD and morale, as illustrated in Fig.7. Each entity represents a variable that we have studied. Each arrow is starting from an entity, E1, and ending with another entity, E2, denotes a positive influence of E1 over E2. Each arrow is the result of

المنارة للاستشارات

www.r

our findings, as reported in the figure's legend. This model shows the overall picture of our findings and how various variables interact with each other. Other relationships might exist, which are not observable from the data collected in this study and are not shown in our model.

As shown in Fig. 7, TD management promotes a culture within organizations in which developers are supported and appreciated for managing their TD. Promoting such a culture not only helps to prevent TD but also has a positive influence on the team morale itself, which ultimately increases developers' morale. On the other hand, the appropriate use of tools and methods (which need to be put in place by TD management) helps to prevent TD and, as a result, reduces the amount of waste of time generated by it. Limiting the amount of working time wasted because of TD has a positive influence on the sense of progress among developers, which increases both the team's morale and developers' morale.

In conclusion, as can be seen in Fig. 7, an appropriate level of TD management leads to a virtuous cycle (C-F-G-I-B2) for which the right culture and TD prevention mechanisms reinforce each other, leading to happier developers.

14.6.Limitations, and Threats to Validity

As empirical research, this study is subject to different threats. We discuss these limitations according to four types of threats to validity suggested by Runeson and Höst [73].

Concerning the *construct validity* [73], there is a concern about using the right operational measures for studying the concept of morale. To mitigate this threat, we decided to follow an approach that relies on capturing different levels of morale by investigating its antecedent factors [18]. As a result, to avoid any misconception, instead of directly asking questions about morale, we asked questions about the influence of TD and its management on a set of factors affecting morale. Also, to avoid misunderstanding the research questions, both the interview and survey questionnaires were tested before data collection. Finally, we recorded all the interviews, and the research team reviewed the transcription of these interviewees to avoid any misinterpretation of the collected data. However, since in this study, we took a holistic approach to examine the influence of TD and its management on morale, future research is needed to provide a more specific understanding of how different types of TD and TD management activities can influence developers' morale.

Furthermore, even if there are other studies investigating the relation between other human factor constructs and productivity, such as, for example, practitioners' happiness [210], satisfaction, and motivation, this study focuses only on the relationship between productivity and morale as a human factor. However, in future studies, it would be interesting to compare our results with other human factors constructs.

This study also uses the estimation of wasted working time due to experiencing TD as a proxy for productivity, but we do acknowledge that lack of waste does not guarantee

المنارة للاستشارات

www.r

productivity by default even if having to spend work time on TD tasks and activities is one obvious impediment towards achieving productivity.

Regarding *internal validity* [73], there is a risk that developers' morale is influenced by uncontrolled factors other than the occurrence and management of TD. To mitigate this risk, at the beginning of each interview, we asked the interviewees to choose specific items from their TD backlog and answer our questions accordingly. However, since we cannot be sure that no other uncontrolled factor has affected developers' morale, future research, preferably in a controlled experimental environment, is needed to address this limitation. Further, in this study, we have analyzed data to find a correlation between specific parameters in the reported data.

Further, we acknowledge that these statistical correlations do not imply causality between the variables. This is because there is a risk that the observed outcomes are affected by other factors that have not been accounted for in our empirical study. The results of this study could potentially be affected by this threat since some of our findings are correlational and potentially also indicate a causal relationship. For example, when we examine the amount of wasted time, correlated with the different statements. By performing this correlation, this validity could potentially have been violated by either finding relationships that are non-existent or missing real relationships that are wrongly deemed non-significant. However, by combining correlation analysis with analytical causality (from e.g., the conducted interviews), causality links can be suggested. Therefore, to mitigate this threat, we triangulated the data by conducting follow-up interviews validating the derived results.

The *external aspect of validity* addresses the possibility of generalizing the findings [73]. The results in this study are based on data from several interviews, a survey, and a longitudinal study. The responses that our respondents gave in the longitudinal part of the study and during the survey and in the interviews might not be representative of the entire developer population. We cannot generalize the results. However, in some phases, we can rely on a good number of participating organizations (7) in different business and application domains. We also provide an online replication package at https://doi.org/10.5281/zenodo.3627885, with information and data organized per phase of the study (as presented in Fig 2), which will facilitate the replication of the study. Furthermore, in surveys, there is always a risk that the sample is biased, and, therefore, a potential threat relates to, for instance, the geographical, cultural, and demographic distribution of response samples.

Finally, since our findings are partially grounded in qualitative interviews, there is a concern about the *reliability* [73] of the findings. We tried to mitigate this threat in different ways. First, we used triangulation both during the data collection and data analysis processes. To increase the reliability of the data collection process, at least two researchers were involved in planning and conducting the interviews (i.e., observer triangulation). Also, to complement our findings from the interviews, we conducted a longitudinal study and asked a group of software professionals to provide their opinions about a set of statements based on our results (i.e., methods triangulation), together with reporting on how much time they wasted. To increase the reliability of the findings

المنارة للاستشارات
www.n

further, even though the qualitative data analysis was performed mainly by the third author, each phase of the data analysis process and its outcomes were monitored and reviewed by the whole research team. Finally, while monitoring the responses to the surveys in the longitudinal study, we realized that one of the respondents had rated all the statements neutrally (i.e., straight-lining). Therefore, we decided to remove the ratings of this respondent to avoid any threat to the validity of data analysis. It must be noted that excluding this respondent affects only the median value of the ratings for ST5.

Their anonymity was assured multiple times to the participants, both by email and in the introduction of the survey. However, we notice that for two statements in Table V, for which a high rating could be perceived negatively from the perspective of interpersonal relations (i.e., ST1 and ST4) the median is '1', while for the more neutral statements the median represents a higher rating: this could mean that developers may have been reluctant to provide answers that might be perceived as 'hostile' to their colleagues. On the other hand, ST5 and ST7 (positive interpersonal relations) are not so extreme in the opposite direction, e.g., showing a very high median, but just 4 and 3, respectively. In conclusion, although a threat exists, we do not deem it to be extremely likely or disruptive. Another limitation is related to the scope of our study. In our model in Fig.7, we show how TD management essentially has only a positive impact on developers' morale and productivity, which in turn increases further the morale. However, this study has not investigated any negative impact of too much management to avoid TD being in place. In fact, industrial processes are routinely asking developers to perform maintenance tasks, refactoring activities, quality measurements might have the opposite effect on morale.

14.7. Conclusion

The presented research aimed to examine the potential impact of technical debt and its management on developers' morale and productivity. To the best of our knowledge, this study is the first study to assess these relationships empirically using a mixed-method approach. The findings from this study make several contributions to the present TD research. First, the results from this study indicate that the occurrence of TD reduces developers' morale since the presence of TD hinders the developers' progress and reduces their confidence. Second, the results imply that proper management of TD increases developers' morale since it enables the developers to perform their tasks better and to improve software quality in the future. Thirdly, the results also suggest that proper TD management leads to a virtuous cycle where the right culture and TD prevention mechanisms reinforce each other, leading to less waste of time, followed by a continuous increase of the developers' morale and productivity. lthough the findings of this study are interesting and encouraging, future research can provide a better in-depth understanding of the relations between the occurrence and management of TD and developers' morale. Especially by using a wider range of data sources and utilizing different psychological measures, future research can perhaps indicate the strength of such impacts and explore any potential differences between development contexts. Further, even if our twofold model illustrating the relationship between TD and morale (fig. 7) is an empirically

المنارة للاستشارات
www.r

grounded model, it can be further validated using, e.g., structural equation modeling with both surveys and software metric data as part of future research.

Acknowledgment

We thank all the anonymous interviewees and survey respondents for their contribution to this work.

المنارة للاستشارات

www.r

15. Technical Debt Tracking: Current State of Practice

In this paper, we investigate the state of practice in several companies, to understand what the cost is of managing TD, what tools are used to track TD and how a tracking process is introduced in practice.

Large software companies need to support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered if both evolution and maintenance of existing systems are hampered by Technical Debt. Although a lot of theoretical work on Technical Debt has been recently produced, its practical management lacks empirical studies. In this paper, we investigate the state of practice in several companies, to understand what the cost is of managing TD, what tools are used to track TD and how a tracking process is introduced in practice. We combined two phases: a survey, involving 226 respondents from 15 organizations, and an in-depth multiple case-study in three organizations, including 13 interviews and 79 Technical Debt issues. We selected the organizations where Technical Debt was better tracked, in order to distill best practices. We found that the development time dedicated to managing Technical Debt is substantial (an average of 25% of the overall development), but mostly not systematic: only a few participants (26%) use a tool, and only 7.2% methodically track Technical Debt. We found that the most used and effective tools are currently backlogs and static analyzers. By studying the approaches in the companies participating in the case-study, we report how companies start tracking Technical Debt and what are the initial benefits and challenges. Finally, we propose a Strategic Adoption Model for the introduction of tracking Technical Debt in software organizations.

This chapter has been published as:
Technical Debt Tracking: Current State of Practice – A Survey and Multiple Case-Study in 15 large organizations
A. Martini, T. Besker, and J. Bosch
Science of Computer Programming, Volume 163, 1 October 2018, Pages 42-6.

15.1. Introduction

Large software companies need to support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered if both evolution and maintenance of the systems are hampered by Technical Debt.

Technical Debt (TD) has been recently studied in the Software Engineering literature [91], [7], [15], [32]. TD is composed of a debt, which is a sub-optimal technical solution that leads to a short-term benefits as well as to the future payment of an interest, which is the extra-costs due to the presence of TD (for example, slow feature development or low quality) [101]. The *principal* is regarded as the cost of refactoring TD. Although accumulating Technical Debt might prove useful in some cases, in others the interest might largely surpass the short-term gain, for example by causing development crises in the long term [110].

المنارة للاستشارات

www.r

There are several kinds of TD, such as Architecture TD, Testing TD, Source Code TD etc. [7],, depending on what artifact and on what level of abstraction the sub-optimality has occurred. TD can only partially be measured by static analysis tools; the rest of TD needs to be tracked, otherwise it will be invisible, as outlined in a quadrant by Kruchten et al. [18]. In 2011 Guo et al. proposed an initial portfolio approach, with the creation of TD *items* to be tracked and managed, which was empirically studied [219]. Seaman et al. have identified the theoretical importance of TD as risk assessment tool in decision making [220]. However, current literature does not cover a number of aspects related to TD: how teams manage (and track) TD, what tools are used in practice and how TD management is introduced in large organizations. Finally, current literature lacks a quantification of the effort spent by the practitioners for managing TD.

In this paper, we therefore aim at addressing the following RQs:

RQ1 How much of the software development time is estimated to be employed in managing TD?

It is also important to understand how a TD tracking process is introduced and implemented in large software companies:

RQ2 To what extent are software practitioners familiar with the term Technical Debt?

RQ3 To what extent are software practitioners aware of the TD present in their system?

RQ4 To what extent do software organizations track TD?

RQ5 Is there a difference between individual and collective management of TD?

RQ6 Does the background of the respondents influence the way in which TD is managed?

RQ7 What tools are used to track TD?

RQ8 How do software organizations introduce a TD tracking process?

RQ9 What are the initial benefits and challenges when large organizations start tracking TD?

To shed light on these question, we have conducted a survey in 15 organizations with 226 participants and we have carried out a multiple case-study in three companies that have started tracking TD: in this context, we have interviewed 13 practitioners responsible for tracking TD and analyzed 79 TD items from a pool of 597 improvements. Our findings include the following contributions:

1. The cost of managing TD in large software organizations is substantial and it is estimated to be, on average, 25% of the whole development time
2. We list the tools that are currently used to track TD, and we provide a first assessment of which ones create less management overhead
3. We report the state of practice related to the introduction of a TD management process in 15 Scandinavian organizations
4. We report the lessons learned from three companies that have started tracking Technical Debt: their starting process, the perceived benefits and the challenges.
5. We propose a Strategic Adoption Model for Tracking Technical Debt (SAMTTD), aimed at helping companies in assessing their Technical Debt management process

المنارة للاستشارات

www.m

and in taking decisions on its improvement. The model also defines the next research challenges to be addressed in theory and to be evaluated in practice.

This paper adds new and more in-depth results on the findings reported in a previous paper [98]. In particular, we address new research questions (RQ2, RQ3, RQ5, RQ6, RQ7), while we add new insights related to the relationship between RQ4 and RQ7 (or else, we study how the practitioners' perception of tracking Technical Debt is related to their usage of tools).

The remainder of the paper reports our methodology in section II, the results in section III, and then we discuss the results in section IV, concluding in section V.

15.2. Methodology

For the execution of this study, we aimed at combining different sources of data (source triangulation) and different methodologies (methodology triangulation), in order to obtain reliable results [73]. To fulfill these triangulation strategies, we conducted a *survey* among 226 participants. The different sources included 15 large organizations and different roles, i.e. developers, architects and managers. To complement such quantitative investigation, we followed up with a qualitative, in-depth multiple *case-study* at three of the companies involved in the survey, which have started tracking TD.

Here we conducted *interviews* with 13 employees and we analyzed *documents* including 79 *TD issues* out of a pool of 597 improvements presents at the companies.

15.2.1. Survey

In this study, we have involved 15 software organizations, belonging to eight distinct large software companies. We consider a *large software company* an organization with more than 250 employees. As visible in the descriptive statistics in Table 2, 91.6% of the respondents reported working for an organization bigger than 250 employees.

The remaining 8.6% were consultants from small/medium organizations working on the same systems and projects developed by the large organizations participating in the survey. They can therefore be considered as working in the same context as the other 91.6% of the participants.

Seven out of eight companies developed embedded software, while another one developed software for optimization (company D). The companies are anonymized and named A-H, and the sub-organizations are called B1, B2, F1-F4 and G1-G4.

15.2.1.1. Survey Data Collection

In the first part of the survey, we asked about the participants' background information:

- Software development experience: "< 2 years", "2-5 years", "5-10 years", "> 10 years"
- Role: "Product Manager", "Project Manager", "Software Architect", "Developer", "Tester", "Expert", "Other (Specify)"

المنارة للاستشارات
www.r

- Gender
- Education
- Team size
- Organization size
- Size of their current project in MLOC (Millions of Lines of Code)

In the second part of the survey, we have asked and analyzed the data related to the effort *caused by* several Technical Debt challenges. The challenges were listed as reported in current literature and not as generic "Technical Debt", in order to make sure that the respondents did not misinterpret the question.

The different kinds of TD are reported in TABLE I. , together with their scientific names and the related academic source. This assured that a better construct validity of our survey was achieved, as we reduced the subjectivity of the respondents interpreting "Technical Debt".

TABLE 1 - KINDS OF TECHNICAL DEBT RECOGNIZED IN [15], [143]

Survey entries	**Source and literature term**
Lack or low quality of testing	Test Debt [15]
Low code quality	Source Code Debt [15]
Lack or low quality of requirement	Requirement Debt [15]
Lack or low quality of documentation	Documentation Debt [15]
Dependency violations	Architecture Debt [15], [143]
Complex architectural design	Architecture Debt [15], [143]
Too many different patterns and policies	Architecture Debt [15], [143]
Dependencies to external resources/software	Architecture Debt [15], [143]
Lack of reusability in design	Architecture [15], [143]
Uneasy/Tensed social interactions between different stakeholders	Social Debt [15], [143]
Lack of adequate environment and infrastructure during development	Infrastructure Debt [15]

It is important to notice that the details and the results from the questions in the second part of the survey are not included in this paper, as the data has been used to cover a different scope and to answer different questions related to Technical Debt in another work [144]. Therefore, the only questions overlapping between the papers are the ones related to the background of the respondents.

المنارة للاستشارات
www.r

In the third part of the survey, we asked the following questions, some of which can directly be mapped to the RQs. Some of the following the questions are rather statements: in such case, we have asked the agreement of the participants to such proposition.

Q1: “How much of the overall development effort is usually spent on TD management activities?”

Q2: “How familiar are you with the term "Technical Debt"?”

Q3: “I am aware of how much Technical Debt we have in our system”

Q4: “All team members are aware of the level of Technical Debt in our system”

Q5: “I track (using tools, documentation, etc.) Technical Debt in our system”

Q6: “All team members participate in tracking Technical Debt in our system”

Q7: “I have access to the output of the tracking of the Technical Debt in our system”

Q8: “All team members have access to the output of Technical Debt in our system”

Q9: “If you track Technical Debt in your project, what kind of tool(s) do you use?”

The formulation of Q1 was slightly different, as we did not mention “TD” but we referred to the challenges mentioned in the second part of the survey (see TABLE I.). However, we use the formulation in Q1 in the rest of the paper for the sake of readability.

After question Q2, the survey included the following definition of Technical Debt:

*“*Technical *Debt (TD) is constituted of non-optimal code or other artifacts related to software development that give short-term benefits, but cause a long-term extra cost during the software lifecycle.”*.

This definition has been operationalized based on the explanations and definitions given by Cunningham [91], McConnell (presentation given at the workshop at ICSE 2013 [221]). We could not include the most recent one from the dedicated Dagstuhl seminar [4], as it was held after the survey was conducted. However, the difference between our definition and the one given in Dagstuhl does not seem very distant, as visible below:

“In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability”

In our definition, we omitted the second part of the Dagstuhl definition. However, by enumerating the different kinds of TD in the first part of the survey (excluding external qualities from the questionnaire), we can be sure that also the second part of the Dagstuhl definition was also covered, although not explicitly mentioned.

This assures that we provided the participants with a good means to understand what Technical Debt meant when we asked about its management. However, we cannot guarantee that the practitioners actually read and understood the definition.

For question 0, since we wanted to quantify the amount of effort related to TD faced by the companies, we provided a scale including the following options: “<10%”, “10-

المنارة للاستشارات
www.m

20%"… "80-90%" and "I don't know". This question was directly aimed at answering **RQ1**.

For 0, we provided the answers "Not at all familiar", "Slightly familiar", "Moderately familiar", "Very Familiar" and "Extremely Familiar". The answers were mapped on a 5-grades Likert scale, respectively 0-4. This question aimed at directly answering **RQ2**.

For 0-0 we asked the respondents to report their agreement on a 6-grades Likert scale: "strongly disagree", "disagree", "somewhat disagree", and the symmetric scale for agreement. These statements were aimed at answering **RQ3**, **RQ4** and **RQ5**. In particular, we wanted to understand if tracking Technical Debt was an individual activity (by asking the same questions for the individual and about the whole team) and if there was a discrepancy between the awareness of the practitioners and their tracking process.

As for question 0, we asked the participants to report the tools used in a qualitative way (text-box). The input was then post-processed and compiled in the resulting word cloud. This question was used, together with the previous ones, to answer **RQ7**.

15.2.1.2. Survey Data Analysis

First, we analyzed the answers from Q1 in order to understand the magnitude of the estimated effort spent by the respondents on managing TD. We transformed the answers from categorical to numerical: for example, we parsed "<10%" to 5, "10-20%" to 15 and so on. After the calculations, we can re-apply the tolerance interval of +5/-5 and the various means etc. would not change. When calculating the means, we did not consider the "I don't know" answers. However, only a small portion of the answers was of this kind (11.5%).

In order to avoid the bias introduced by different roles answering the questionnaire, we ran a cross-tabulation chi-square test of independence to understand if the role of the participants affected the answers.

The second step was to apply frequency analysis on questions 0-0. In order to do so, we transformed the categorical data to a Likert scale (1-6), where "strongly disagree" was mapped to 1 and "strongly agree" to 6. As for 0, we also reported the grouped answers in three main intervals, "No tracking" (1-2), "Somewhat tracking" (3-4), and "Tracking" (5-6). We used these aggregated intervals only for the last results related to the Adoption Model SAMTTD model.

For some of the results, we used standard *boxplot*. The boxplot is a comprehensive way to visualize various descriptive statistics altogether at a glance. We used this method when we aimed at showing the difference about the distribution of the data with respect to two specific variables. For example, Fig. 3 shows the comparison, with respect to different companies, of the distribution of the management effort: we can compare the medians (the black lines in the middle), but we can also see different percentiles (where most of the answers were concentrated) and outliers. More information on how to read a boxplot can be found at http://flowingdata.com/2008/02/15/how-to-read-and-use-a-box-and-whisker-plot/.

المنارة للاستشارات

www.m

In other cases, we compared the different variables using statistical tests: for example, it seemed interesting to compare how much the respondents were aware of TD with respect to how much they were tracking it. To do so, we performed a number of tests for linear correlation using the tool *R*. Most of the numerical variables did not have a strong linear correlation with each other, with the exception of the answers for 0, 0, 0 and 0: this is not surprising, as if TD is not tracked by an individual, it is probably not tracked by the team, and the output will not be visible to the individual and to the others as well. The Pearson tests for linear correlation gave results from 0.72 up to 0.89 with p-value vastly lower than 0.05. This can be considered a good test for the reliability of the answers. Since these variables all strongly correlate, in the remainder of the paper, when studying different variables, we will only use the "tracking" variable, without considering if the output was available or not.

We also wanted to understand if the results depended on a specific variable. For example, we tested if developers answered differently with respect to architects or managers. We thus ran several chi-squared tests of independence between the background variables of the participants and their answers related to questions 0-0, to answer RQ6. For example, we wanted to know if the familiarity, awareness, tracking, etc. of the respondents would depend on their background, e.g. by their affiliation to a company, to their education, etc. This analysis was done to answer **RQ6.**

We finally analyzed the qualitative answers from 0, in order to better understand the results answering **RQ7**. We selected the answers where the respondents reported that tools were explicitly used (61/226, 27% of the respondents), and we compared the respective levels of awareness, tracking, and familiarity. This was done to better understand what the respondents meant by "tracking". We also created a word-cloud representation of the qualitative answers for Q10: this, we found, could represent quite well which tools were the most used and in what way. To do so, we processed the qualitative data removing terms that would appear in the word-cloud but would not make sense from the tool point of view, for example "code" and "technical debt". Finally, from the coding of the qualitative answers, we could also identify the frequencies of the used tools. To do so, we manually coded the 61 answers in the following six categories:

- *Comments*: these are usually "TODO" comments, left by the developers in the code or other artifacts. These are useful for the developers to know that something is left to do, but it does not imply a systematic monitoring of the TD reported in the comments.
- *Documentation*: from the qualitative answers, this represents a text or spreadsheet where issues are listed and explained in a semi-systematic format. Another example could be a wiki. However, such documentation is different from a backlog as it is more difficult to monitor and it does not use a specific technology to manage and perform operations on the backlog.
- *Issues*: using the same ticket system for bug fixing, but usually down-prioritizing the issues related to Technical Debt.

المنارة للاستشارات

www.r

- *Backlog*: this is either a dedicated backlog for TD issues, or the usual feature backlog were TD items are mixed with features. This practice usually involves a technology such as project management tools.
- *Static analyzer*: these are tools such as SonarQube, SonarGraph, Klockwork, etc. used to analyze the source code in search for Technical Debt. In a few cases, respondents report that they built their own metrics tools. These tools usually check (language-specific) rules or patterns that can warn the developers on the presence of TD. These tools are used as trackers by the developers, with the limitation that they only cover part of the TD.
- *Lint*: they are also static analyzers, but are more used to find potential bugs and security issues rather than technical debt.
- *Test coverage*: some of the respondents measure test coverage and they consider a low test coverage as presence of test debt.

15.2.2. Multiple case-study

In order to better understand to what extent companies tracked TD (**RQ4**) and how the tracking process was introduced (**RQ8-9**), we conducted a multiple case-study, investigating some of the companies involved in the survey. We interviewed 13 employees from cases B1 (project manager, system architect and two developers), F1 (three software architects responsible for TD management in three different teams) and F4 (two system architects, two project managers and two developers). In particular, in order to understand what was considered as "good tracking", we had the opportunities to interview the participants, belonging to company F1, who answered "strongly agree" (the highest level of tracking) to question Q5. This gave us an idea of what was considered as current best practices for tracking TD. In order to support the interviews, we also analyzed 79 out of 597 TD backlogs items used for tracking improvements (and thus including TD items) in company B1, F1 and F4.

15.2.2.1. Interviews

Data Collection

The interview questions were designed to cover taxonomies we found in the pre-study concerning the *reason for initiation*, the *activities* within the TD management process and also the *process implementation*. All interviews were audio-recorded, and the results of the interviews were organized by different questions and activities for later analysis.

We formulated the interview questions in three sections:

- The *first* section contains questions about the profile of the interviewees and their companies.
- The *second* section focused the questions on the initialization of the process for managing TD: "What was the main reason for implementing a TD management process?", "Who decided that the process should be implemented?", "What negative effects did you experience in your system due to TD?" (**RQ8**).

المنارة للاستشارات
www.r

- In the *third* section, we asked about the outcome of the implemented process (**RQ4**) and how the companies experienced the implementation of the process in terms of the most obvious benefits and challenges (**RQ9**).

Data Analysis

The data analysis used an inductive approach based on open coding [222] . We were looking for activities related to the introduction of a TD management process in the company. For this purpose, we followed the points in [223], which is a well-known study on change management in software engineering. The data were coded using a Qualitative Data Analysis (QDA) software tool called Atlas.ti. Such tool gives support to keep track of the links between taxonomies, codes, and quotations. Based on the taxonomies, we developed a coding scheme that contains a corresponding set of codes and sub-codes. Fig. 1 shows an example of our code hierarchy and how the codes were mapped to the taxonomy: the graph is part of the overall data collection model (not completely displayed here for space limitations).

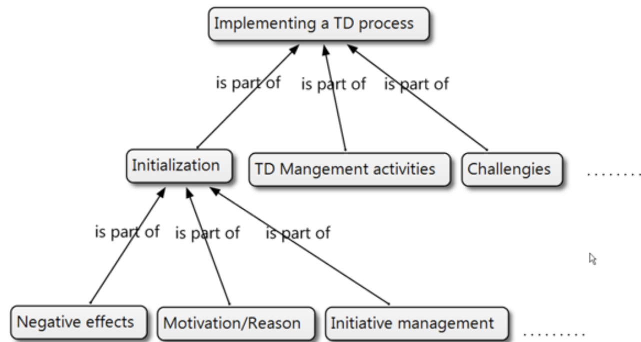


Figure 1. The coding process

As an example of how the coding was conducted, we present a quotation from one of the interviewees which was mapped to the *Motivation* sub-code: *"…we realized that for each and every release it took much time correcting or fixing problems with additional patches and it took more and more time adding new features on top of the system*".

15.2.2.2. Document Analysis

In order to gain more evidence on how the companies were tracking TD (**RQ4**), we investigated the existing documentation. Also, we had access to the TD backlogs of the studied teams: 26 items in the organization B1, 451 items in F1 and 20 items in F4. We analyzed the TD items' fields, values and how they were ranked. We did not analyze all items in company F1, as 451 items included also improvements that were not TD. We randomly selected 30 items that corresponded to the definition of TD, we analyzed them and then we tested our assumptions by randomly looking at other items in the backlog. We used the backlogs in the interviews (see previous section) in order to ask follow-up questions to the participants. In addition, we analyzed the documentation that was created by the organizations to explain TD to the users of the tracking process.

المنارة للاستشارات

www.r

15.3. Results

15.3.1. Demographics and background of the respondents

In total, we obtained 226 complete answers. The total respondents were 259, which gives us a completion rate of 87%.

We aimed at having a similar number of respondents from each organization (Fig. 2). The participants were almost all experienced practitioners, since 156 respondents (69%) had more than 10 years of experience, while only 8 (3.5%) had less than two years of experience (the remaining 62, 27.5%, had between two and 10 years of experience). Several roles participated in the survey: 37 managers (16%), 52 software architects (23%), 105 developers (46%) seven testers (2.65 %), 14 experts (5.75%) and 9 system engineers (4%) completed the survey.

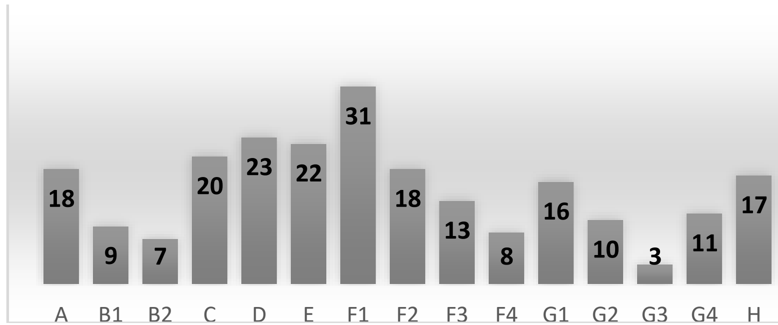


Figure 2. Number of participants per organization

المنارة للاستشارات
www.r

TABLE 2. BACKGROUND DATA RELATED TO THE RESPONDENTS, WITH THE PERCENTAGE, THE NUMBER OF RESPONDENTS AND THE RELATIVE DISTRIBUTION

Option	%	Resp.	Distribution
Practitioners' Experience			
< 2 years	3.50%	8	
2 - 5 years	8.80%	20	
5 - 10 years	18.60%	42	
> 10 years	69.00%	156	
Gender			
Female	7.10%	16	
Male	92.00%	208	
Other / Don't want to	0.90%	2	
Education			
No University education	7.10%	16	
Bachelor degree	24.80%	56	
Master degree	58.00%	131	
Ph.D. degree	4.40%	10	
Other:	5.80%	13	
Team Size			
1-5 team members	21.70%	49	
6-10 team members	38.10%	86	
11-20 team members	15.50%	35	
21-40 team members	5.80%	13	
> 40 team members	19.00%	43	
Organization Size			
< 50 employees	1.30%	3	
51-250 employees	7.10%	16	
251-1000 employees	15.00%	34	
1001-5000 employees	14.60%	33	
>5000 employees	61.90%	140	
Age of current system			
< 2 years	10.20%	23	
2-5 years	20.80%	47	
5-10 years	33.20%	75	
10-20 years	29.60%	67	
>20 years	6.20%	14	

As visible in Table 2, we can infer the following characteristics on the studied sample:

- *Experience*: most of the respondents had more than two years of experience, while 69% of them had more than 10 years of experience. The estimations can therefore be considered quite reliable, as they are made by expert practitioners used to estimate their work.
- *Education*: most of the respondents have a Bachelor or Master degree. The level of education is therefore quite high. However, the sample do not include many practitioners involved in research projects.
- *Team size*: although many of the teams are small (1-10 members), the sample includes a substantial number of respondents working in large teams as well.

المنارة للاستشارات

www.r

- *Organization size*: as mentioned in the analysis made in section 8.2.1, the organization of the respondents is large. This was chosen by design: we wanted to restrict our results to large organizations. This pose a limitation on our study: we cannot generalize these results to small organizations.
- *Age of the current system*: the distribution of the different systems is quite even, as the sample covers almost equally all the different phases of the system. This raises the degree of generalizability of our results, as it assures that our data covers both "young" and "old" systems.

15.3.2. Estimation of management cost of TD (RQ1)

First, we report the answers to Q1 from the survey. In Fig. 3 we show the distribution of the respondents with respect to the different levels of estimated effort that was reported. By picking the middle values, as explained in the methodology section (e.g. 10-20% was transformed in 15), we calculated that the average cost of managing the TD was estimated by 215 respondents to be 25.9 % with a median of 25 % of the whole development time.

From the results, we can see how most of the respondents answered between 0 and 40 %, while half of them are between 10 and 30%. However, some respondents report to spend more than 40% of their time managing TD.

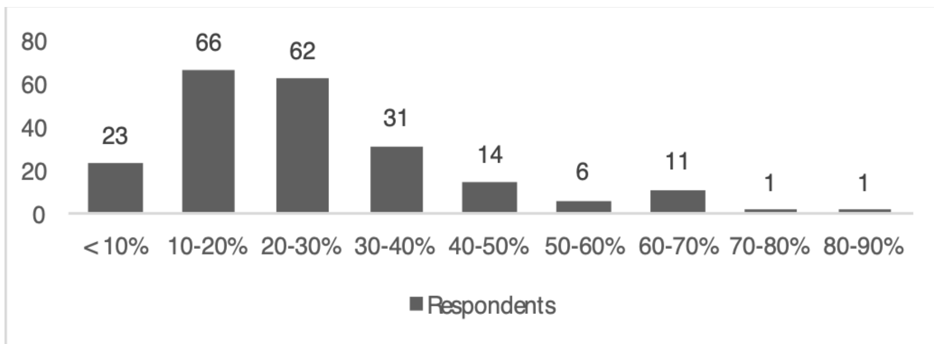


Figure 3. Distribution of respondents for Q1: "How much of the overall development effort is usually spent on TD management activities?"

Looking at the comparison of medians (bold lines) and percentiles among the companies (boxplot in Fig. 4), we cannot see a big difference in how the respondents answered, apart for the slight difference for E, F1 and F3. This means that the amount of time spent to manage TD is quite not dependent on the organization.

المنارة للاستشارات

www.r

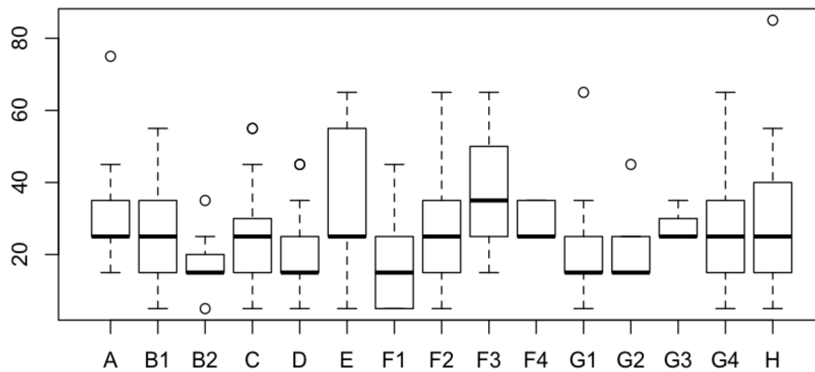


Figure 4. Comparison of companies with respect to Q1:
"How much of the overall development effort is usually spent on TD management activities?"

A chi-square test of independence, aggregating the intervals over 50% in the same category (the lack of values would have invalidated the chi-square test), yielded a p-value of 0.144, so we could not reject the hypothesis that the role of the respondents would influence their answer. This means that the answers did not vary significantly across the roles, contrarily to what one might expect, considering different views and experiences of different roles in the organizations.

15.3.3. Familiarity with the term "Technical Debt" (RQ2)

The respondents seem to be, in total, moderately familiar with the term Technical Debt. The mean is 2.26, while the median is 2. From the graph below we can see how there are more respondents who are *very familiar* with respect to the other ones.

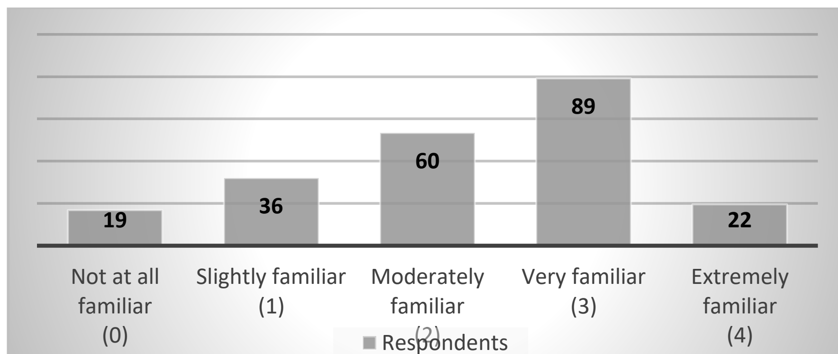


Figure 5. Distribution of respondents according to their answers to Q2:
"How familiar are you with the term "Technical Debt"?"

From the comparison among the companies, we can see how they are mostly on the same level: F4 is above all the rest, while the organizations B2 and G4 are not very familiar with the TD concept. However, since we did not have access to the practitioners working

المنارة للاستشارات

www.r

in these two organizations, we cannot tell what was the cause of this lack of familiarity. We omit the test of independence, as the results are clearly visible in Fig. 6.

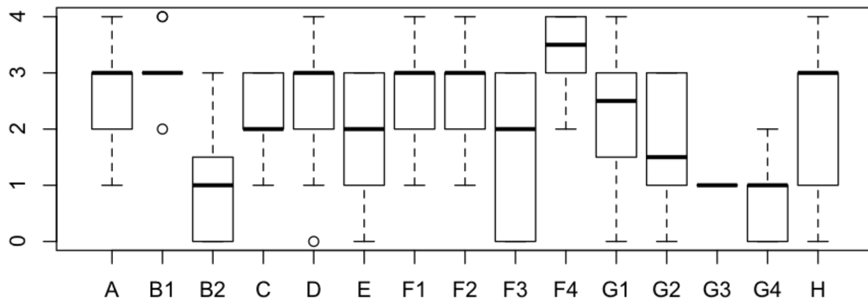


Figure 6. Level of Familiarity with the term Technical Debt for each organization (answering Q2: “How familiar are you with the term "Technical Debt"?”)

15.3.4. Awareness of Technical Debt present in the system (RQ3 and RQ5)

When assessing the level of awareness of the TD present in their system, the respondents, on average, somewhat agree that they are aware of how much TD they have in their system (mean = 3.69, median = 4). Almost half of them (45%) *somewhat agree*, while only 21% feel more confident (they *agree* or *strongly agree*) and the remaining 32% *disagree* or *somewhat disagree*. Only 3% of the respondents were not aware of TD.

On the other hand, the practitioners seemed less convinced that the whole team would be aware of how much TD is present in the system. Here the mean is 2.8, while the median is 3, both close to a mild disagreement. The chi-square test of independence confirmed that the distributions are not dependent, with a p-value < 2.2e-16.

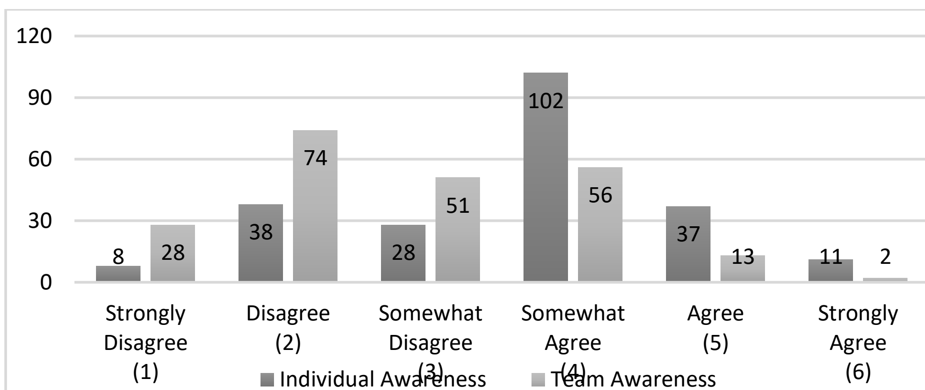


Figure 7. Comparison of answers for Q3: “I am aware of how much Technical Debt we have in our system” (Individual Awareness) and Q4: “All team members are aware of the level of Technical Debt in our system” (Team Awareness)

المنارة للاستشارات

www.r

For what concern the different companies, they are quite aligned on the awareness among each other. Once again, B2 seems to have a somewhat less level of awareness. The results suggest that belonging to one or the other organization would not have an impact on the level of awareness of their employees.

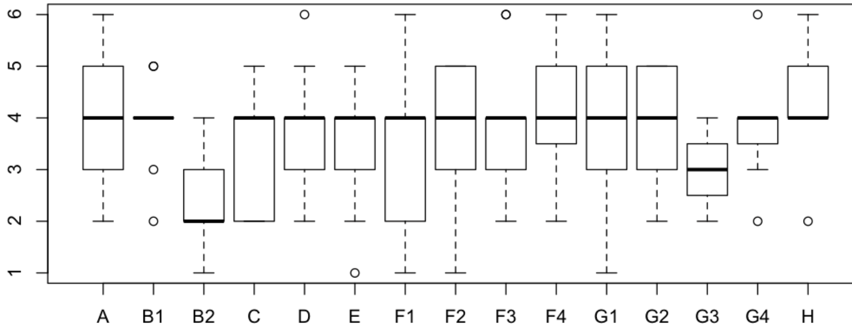


Figure 8. Distribution of answers with respect to Q3: “I am aware of how much Technical Debt we have in our system”. 1-6 correspond to “strongly disagree” to “strongly agree”.

15.3.5. Tracking Technical Debt (RQ4)

In this section, we report the results from Q5: “I track (using tools, documentation, etc.) Technical Debt in our system”. The average tracking level, reported by 219 respondents, is 2.3 with a median of 2. On the team level, it seems to be just slightly worse, as shown in Fig. 9 and discussed below.

Based on the results from a chi-square test of independence between the role and the tracking level, we could not reject the hypothesis (p-value 0.63) that the role of the respondents would influence their answer with respect to tracking. In Fig. 10 we show the comparison among different companies. We can see how the different companies answered in a similar way, apart from company F4 and partly company D. However, the test for independence we did not show any significant relationship between the variable company and the answer given in the survey with respect to Q5 (tracking TD).

Finally, there is very little difference between tracking on an individual (Q5) or team level (Q6). Only some of the individuals track TD more than the rest of their team. This is strongly confirmed by a Wilcoxon test, which rejected the null hypothesis (p-value = 2.008e-05) that the difference in the two paired distributions are given by chance. In other words, the same participant answered very similarly when asked Q5 and Q6, and this is not because of randomness, which means: if someone in the team tracks TD, it is very probable that the whole team is involved in the tracking.

Finally, as observed in the methodology section, the results from Q7 and Q8 (related to whom, in the team, has access to the outcome of TD tracking) very strongly correlated with the answers to Q5, so we do not report the exact results here. In other words, this means that the respondents who track TD have also access to its output (e.g. backlogs, dashboards, etc.)

المنارة للاستشارات

www.r

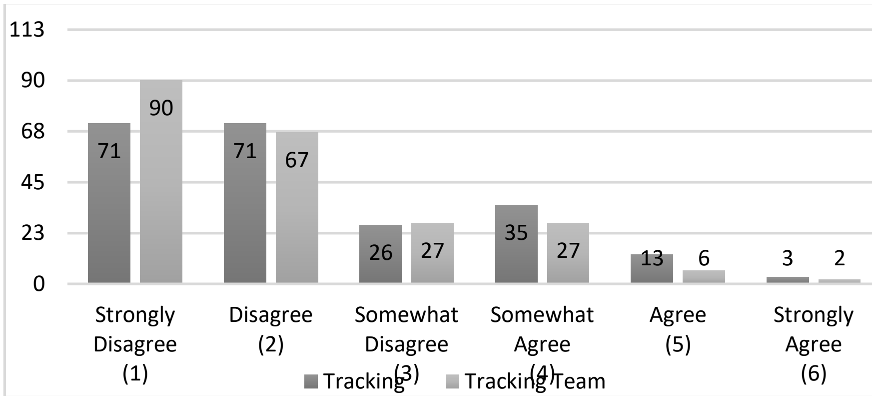


Figure 9. Distribution of answers related to Q5: "I track (using tools, documentation, etc.) Technical Debt in our system" and Q6: "All team members participate in tracking Technical Debt in our system"

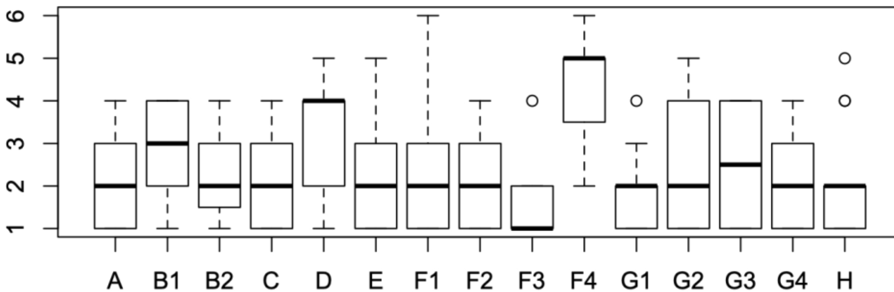


Figure 10. Distribution of answers for Q5: "I track (using tools, documentation, etc.) Technical Debt in our system"

15.3.6. Influence of the background of respondents on the management of TD (RQ6)

We have partly answered RQ6: "Does the background of the respondents influence the way in which TD is managed?" in the previous sections, especially with respect to the variables *roles* and *organizations*. However, we had several other variables in the background section, and we investigated if any of those variables would help understanding what cause a more or less mature TD tracking. To answer this question, we ran several statistical chi-squared tests of independence between the background variables (education, team size, etc.) and the variables of interest (familiarity, awareness and tracking of TD). However, none of the statistical tests yielded a significant answer: technically, we could not reject any hypotheses for which the answers were dependent on the background of the respondents. Since the results would include several combinations of p-values that would not add any meaning to the manuscript, we decided to omit such table.

In conclusion, the management of TD depends on some factors that have not been captured by the surveyed background variables. However, in the next sections, we provide

المنارة للاستشارات

www.r

some answers that could not be found in the quantitative data, but seem to be related to the historical and social context where the participants work. More information is given in sections 8.3.8 and 8.3.9.

15.3.7. Tools used to track Technical Debt (RQ7)

In this section, we analyzed if the respondents, who used some tools to track TD, were also more aware of TD or tracked it more than the other ones. To do so, we took in consideration only the answers from the 61 participants (27%) that answered the qualitative question Q9 (specifying what tools they used). We also report the mean values for the questions 0 and 0: we compared the answers of the participants who used a tool with the ones who did not. We found the results in TABLE II. : it seems that, indeed, if the participants used a tool to track TD, then they would report a high perception of tracking TD. A chi-squared test of independence confirms a strong difference in the distribution of the answers (p-value < 2.2e-16), strongly confirming this claim. However, more surprisingly, their perception of the level of awareness of how much TD is present in the system, would only slightly change. This is confirmed by a chi-squared test of independence (p-value of 0.59), which did not show any difference in the distribution of the answers between the participants using a tool or not. Very similar results were found at the team tracking level, so we do not report them in the table below.

Given the high difference in tracking between the respondents who claimed to use a tool and the ones who did not, we can safely claim that the respondents tracking TD also use a tool. Although this seems straightforward, there is also the risk, in a survey, that some respondents just skip a question that is not mandatory. However, this result confirms that we captured most of the respondents' answers related to the tool that they used. The respondents who did not input an answer for the tool, also most probably don't use any tool, since they have in general a much lower level of tracking. Therefore, we can further validate the result that only 26% of the participants used a tool to track TD. This is also important for the reliability of our results related to the SAMTTD model explained in the next sections.

From the qualitative data, we could also report what tools were used in practice. After removing some of the words that would just create noise (such as "Technical Debt", see methodology section for more details), we obtained the following word-cloud, which shows the distribution of tools used among the respondents:

المنارة للاستشارات

www.r

Figure 11. Word cloud of the tool used by the participants to track TD

By codifying the qualitative answers in *comments, documentation, issues, backlog, static analyzer, lint* and *test coverage*, we can also analyze the frequencies. We can see how the tool that was mostly used is a backlog (dedicated to TD or the same used for feature development), followed by documentation, static analyzers and issue trackers (Fig. 12).

Figure 12. Number of participants using a specific kind of tool

المنارة للاستشارات
www.r

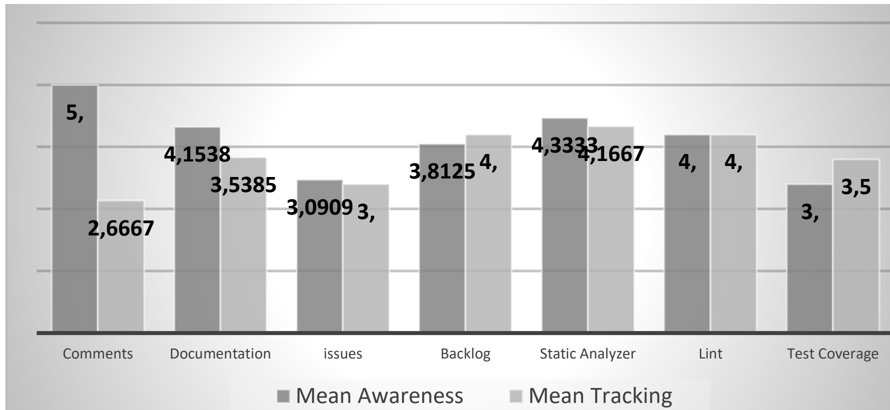


Figure 13. Level of TD tracking and awareness related to the user of each kind of tool

We then analyzed the mean of the respondents for *Awareness* and *Tracking* levels (Fig. 13) with respect to the different kinds of tools. From these results, we can infer that there is not a huge difference among the different used techniques. On the other hand, by analyzing the kind of used tool with respect to the amount of effort spent on management activities (Fig. 14), we can see a quite clear difference. Although this difference could not be statistically tested (the chi-square tests did not report significant difference, but this could be due to the small sample), backlog and static analyzers are the ones that seem to create less overhead.

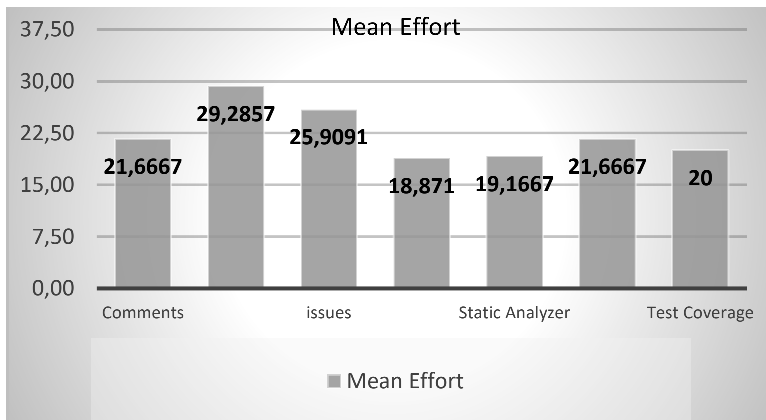


Figure 14. Mean of management effort for each kind of tool

In conclusion, the following considerations on the tools can be made:

- Comments in the code help awareness, but they are not considered tracking and they are used by just 1% of the respondents. This is probably because they are not used in a list that can be monitored by the team outside the code.

المنارة للاستشارات
www.r

- Documentation increases TD awareness, but it is not considered as a high level of tracking and it has the highest overhead. The main tool used here were Microsoft Excel or Word. We can infer that this practice is not recommendable in comparison with the other ones.
- Using a bug system for tracking TD is not considered as contributing to a better level of awareness or tracking compared to the other techniques, and it has a slightly higher overhead. We would infer that this is also not the best way of tracking TD.
- Backlogs, static analyzers and "lint" programs all increase the tracking level, but we cannot see a big difference (although static code analyzers seem to contribute better to the participants' awareness). They are also the ones with the least overhead. They therefore seem to be considered the best practices at the moment to track TD.
- Backlogs are the most used tool among the participants. In particular, the most used backlog tools are Jira, Hansoft and Excel.
- Test coverage does not seem to contribute too much to the awareness and tracking level, although it does not involve much overhead. This might be due to the fact that test coverage is related to only a small part of TD.

15.3.8. Why and how do companies start tracking TD? (RQ3)

First, we report *why* the companies decided to start tracking TD, or else their *motivation*. Then, we found that the *preparation* activity was critical to start tracking TD, and we therefore report the main steps involved in this practice.

15.3.8.1. Motivation for start tracking TD

The main reasons behind the start of tracking TD were related to experiencing the *interest* of TD, or else: *too many bugs* to fix, *decreased feature development*, *performance issues*:

*"*Because *we realized that for each and every release it took much time correcting or fixing problem with additional patches and it took more and more time adding new features on top of the system. [...] The system became more and more inefficient".* These statements confirm our previous results [110], as one of the architects also mentioned: *"After some time the TD was increasing and we had a crisis situation".*

In other words, the main motivation was related to the negative impact experienced by the practitioners, or else the perception of the interest associated to the TD.

15.3.8.2. Preparation of the tracking process

From the cases investigated, it was clear that adopting a TD tracking process requires some initial activities and time to implement the process. From B1, we understood that they *"Have done this for 1.5 years more or less, switching from reactive to more proactive. It's a better information about the status of* the *system."* The *preparation*

المنارة للاستشارات

www.r

includes the following aspects. Although we used [73] to code these results, we prefer to report them in a way that is more readable:

- Initiative – in all the three cases, the tracking process started from an individual initiative. A manager, a system architect, an experienced developer, etc. In other words, tracking TD requires a *champion* in the sub-organization, who is aware of TD and is willing to promote the adoption of the practices.
- Budget – tracking TD needs both an initial effort and a continuous effort. Company B1 started with 150 hours, in the beginning, for a development unit (i.e. a sub-organization responsible for a sub-system, which includes a few teams): however, this was *"ok just to start the backlog, but not to go in-depth investigation"*. The continuous time allocated to tracking TD varied across our cases: it ranged from 10% (company F) to 30% (company A). The cases also show how the continuous allocation of resources to manage TD could be dynamic, and varying according to newly identified items, as suggested for Architectural TD in [135].
- Management involvement – although the initiative can start from anyone in the organization, tracking TD requires an initial and a continuous investment (budget). This entails the need of involving a manager who understands the importance of TD and who can grant a budget for this activity.
- Benefits – as the previous point entails, there is a need, for the management, to understand the benefits of tracking TD given the initial and continuous budget allocation. Such benefits need to be communicated and continuously evaluated in order to justify such investment.
- Measurement set up – according to company B1, an amount of time is needed to set up measurements (e.g. complexity) and TD identification (static code analyzers). In other companies, such as F1, we found that a developer set up a specific analysis tool to measure complexity and bug density: this activity was supported by a team dedicated to the measurements in the organization.
- Explanation and alignment – the *Champion* for the TD tracking activity needs to communicate well to the teams what TD is and what needs to be reported (to avoid overhead). The interviewees mentioned that they conducted first workshop for explaining TD and its tracking, and they also produced some documentation. It is also important to have a *validation* workshop, where the teams bring up some TD issues in order to align their understanding with the main TD concepts such as *Principal* and *Interest*.
- Appointing of a Sub-System TD Responsible (SSTR) – TD tracking needs responsible across the organization who take the initiative to support the tracking process. In all the studied cases, the responsibility for collecting and maintaining a list of TD issues were chosen as experienced developers on a given sub-system. The sub-system TD responsible, however, needs to be supported by the knowledge of the teams when tracking the issues, as different practitioners have better and more detailed views of different parts of the system.

المنارة للاستشارات

www.r

- Breaking down and distributing TD items – The SSRT needs to allocate the TD items to the teams according to their competences and their responsibilities with respect to the system. Architecture items were explicitly appointed to an experienced developer to be analyzed and estimated.
- Communication of TD to management – Once the first TD backlog is prepared, it was communicated to a manager, connected to the evaluated (sub-)system. This was supposed to show the management the risk associated to such system due to TD.

In summary, quite a few activities are necessary to set up a TD tracking process: this requires the organizations to take the initial decision of allocating some budget to TD tracking.

15.3.9. What are the benefits and challenges of tracking TD? (RQ4)

15.3.9.1. Benefits

When we evaluated the tracking process together with the teams, they mention several benefits of tracking TD. The backlog gave them a long-term perspective, not only the short-term one given by the feature backlog. The respondents did not think that the TD backlog was hard to maintain. This is supported by the lower management overhead reported in the survey with respect to the other practices.

One of the architects in organization F4 mentioned that, after an important architectural TD item was refactored: *"The evidence was visible in the next release with positive impact when adding new features* on *top of the one we fixed. Easier to add and no side effect, cleaner architecture."*. According to the project manager interviewed in company B1, the initiative was overall successful, but it needed to be continuously supported, to be really effective: *"Yes it was worth it but it is important to follow it up now and to make sure that parts of the list are done [refactored]."*

Another benefit reported by the architect in company B1, the initiative and the weekly meeting promoted a discussion with teams working on other systems, for example when an issue on the interfaces was revealed. The same architect reported that the TD backlog was a great way to receive feedback from the developers, as it made clear what, according to them, was important to refactor.

One of the interviewed SSTR and a developer mentioned that focusing on TD was important in order to *"zoom out"* from their daily work and it was an opportunity to check the system with a broader perspective. Finally, all the architects mentioned that, by using the tracking process, they learned issues that were not known before.

15.3.9.2. Challenges

Although several benefits were mentioned by the respondents, some issues with the current approaches were also reported. The most important one was the acceptance, from the managers, of the need for refactoring. Even with the list updated, the information

المنارة للاستشارات

www.r

about the risk and benefits of performing a refactoring was not always clear to the managers. This meant that, especially for large TD items, it was difficult to receive the needed budget for TD repayment.

One of the major problems in starting tracking TD was that the first step needed a substantial amount of effort in order to collect all the existing items. Although this effort would be one-time only, in some teams the managers would not concede the necessary budget. A challenge mentioned by all the participants was that the refactoring became more difficult to be prioritized and completely repaid when several items and several teams were involved. It required "*double*" the effort to prioritize the item with different managers (who could disagree on the necessity of refactoring) and the coordination of the refactoring was considering quite risky and as a dangerous overhead. For example, TD issues involving interfaces were more time consuming to estimate and prioritize, as they required more discussions involving more stakeholders from more teams.

Another challenge in the prioritization activity was the difficulty to prioritize *among* TD items, especially where an explicit risk/impact value was not calculated. The participants reported that it was generally difficult to show an actual gain from the cost/benefit analysis to the managers, even with a field explicitly represented in the backlog. In general, the intuitive values used for the risk/interest (but usually not including a systematic calculation) were working only sometimes, and more explanations and indicators were required by the managers to accept a costly refactoring.

The respondents mentioned the difficulty to coordinate the different teams in using a standardized process for tracking TD. In some cases, it was difficult to "*make them care*" about reporting TD, while for other teams the TD list was created with enthusiasm.

Finally, the participants mentioned that in some cases the TD backlog itself did not make the TD more convincing for the management to be refactored, but it served for the teams to remember to take care of TD, which would otherwise remain invisible and overlooked.

15.3.10. Strategic Adoption Strategy

As a final result from the combination of the various analyses performed so far, we aggregated the results, and we combined them together with the roadmap related to the current literature on TD. This led to the Strategic Adoption Model for Tracking Technical Debt (SAMTTD, Fig. 6). The first four steps in the model represent the results from the survey on the current state of practice in the companies.

In particular, we used the results from Q4 to create the first step: if the respondents were not familiar with the TD concept, they could be on a higher level. Then, we defined three more levels of TD tracking maturity. To discern between the different levels, we mapped practices that we found used or not and that correlated with different levels of tracking (e.g. the usage of a tool). We additionally used the results from the interviews, where it was clear what different practices were introduced to track TD.

- *Unaware*: there is no awareness of what Technical Debt is and therefore how to manage it. According to our survey data, only 8.4 % of the participants are in

المنارة للاستشارات

www.r

this stage. This datum is related to the respondents that answered “Not familiar at all” with the term Technical Debt, as visible in Fig. 5.

- *No tracking*: in this stage, the software engineers are aware of the TD metaphor and there is a general understanding of the negative effects brought by having TD in the system, but there is no initiative to track TD, which remains invisible. Around 65.6% of the respondents report to be on this level, by (strongly) disagreeing about tracking TD. The % was calculated by counting the total answers minus the answers from Q4, counted previously as the *unaware* respondents, and the ones who use tools, counted in the next levels (26%). Therefore, this yielded 100-8.4-26 = 65.6.
- *Ad-hoc*: In this stage, the software engineers are aware of what TD is and some of the individuals have started tracking TD on their own. This makes the TD management process ad-hoc, since, without a dedicated budget, such individuals use what is available, in terms of tools and processes, for other activities. For example, according to the qualitative answers related to Q3, the sprint or product backlog, a common issue tracker or a simple excel spreadsheet can be used for the purpose of tracking TD. Static analysis tools might be in use, but are limited to the individual usage. According to the survey, approximately 26% of the respondents are at least on this stage (61 participants, 26%, were using tools, see section 8.3.6). However, from these ones, we need to take away around 7 % that we place on the next level (see point). In total, we therefore report around 19% of respondents on this level.
- *Manual tracking*: the company in this level has acknowledged the importance of tracking TD also on a management level (see Preparation section). Therefore, there is a budget generically associated with the management of TD. This amount usually ranges between 10% and 30%. According to the document analysis of the TD items from the case-study, a specific backlog and documentation related to TD is necessary, with TD-specific values useful to analyze the principal and the risk/interest. The TD is understood by the participants, who have been instructed by a responsible for the process (see Preparation). There is an iterative process in place to monitor TD (identify, estimate, prioritize and repay it), and such process is subjected to continuous improvement. 7.2 % of the respondents are on this stage, actively tracking TD. This is the maximum level achieved by the companies, as confirmed by the interviewees. This amount can be obtained when taking in consideration the respondents who answered “Agree” or “Strongly Agree” to Q5 (see Fig. 9).

We do not have evidence that companies have better processes and tools in place. However, based on current literature on TD [15] and on related work on change management [223], we hypothesize future maturity steps that can be reached by the companies when the results of research would be put in place. We identify the following three steps:

- *Measured*: in this stage, identification tools for TD are in place, for example the use of the tool SonarQube for source code TD (such as McCabe complexity) or,

المنارة للاستشارات

www.r

for example, dependency checkers on the architecture level (as reported in company F1). The measurement of the interest is also in place, e.g. there are indicators that show the amount of interest paid or predicted if the refactoring is not conducted. Such tools are not employed in practice yet, and should be integrated in order to provide overall indicators to be provide help to the stakeholders to estimate and prioritize TD. The authors of this paper are actively working on introducing such tools and indicators, as explained in our recent work [54].

- *Institutionalized*: according to change management [223], a process is mature when is spread and standardized across the whole organization. This would allow an aligned prioritization of TD across the system. This would also allow the practitioners to plan the allocation of resources according to the quality of the (sub-)systems in order to plan for the lifecycle of the product. As an example, the reader can consider a team who needs to build a feature on a sub-system developed by other teams: knowing how much TD is present in such sub-system, would allow the team to estimate if a refactoring is needed or the lead time for the features to reach the customer.
- *Fully automated*: In this stage, the decisions on the refactoring are completely data-driven, making use of statistics collected on historical data or by benchmarking the system against a collection of reference systems. For this purpose, however, the previous steps are necessary.

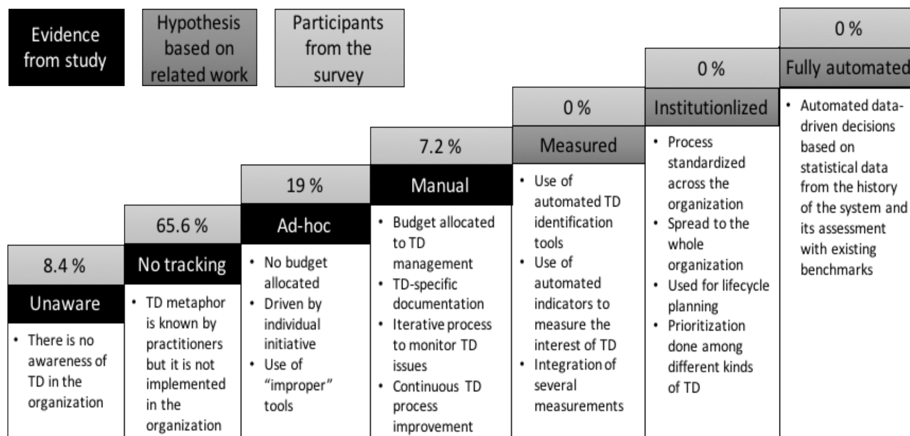


Figure 15. The Strategic Adoption Model for tracking Technical Debt: the main milestones and the state of practice (% of respondents per category)

15.4. Discussion

The combination of data from 226 participants in 15 large software organizations with the in-depth case-study, provided an overall picture of the current state of practice with respect to TD tracking. In this section, we discuss the contributions in this manuscript,

المنارة للاستشارات
www.r

with respect to practitioners and researchers, we compare our results with existing literature, and we report limitations and threats to validity related to our study.

15.4.1. Current state of practice of tracking TD and implications for practitioners and researchers

The results related to RQ1 tell us that software companies spend, on average, around 25% of their development time on TD management activities. The boxplots (Fig. 4 and Fig. 10) show some consistency in the companies: the medians range between 15 and 25% of effort. The 50-percentile also shows some consistency in the answers, but there we can find some variance as well: different organizations and individuals dedicate divergent amounts of time to TD management. We could find some differences in the approaches used by the participants, which seems promising: for example, the usage of backlog and code static analyzers appear to be related to less management overhead. However, considering the quantitative analysis, we cannot infer that any background variable related to the respondents would have a significant impact on the overall management overhead.

The results related to RQ2, RQ3 and RQ4 tell us that only a few employees do not know what TD is (around 8%). However, despite the familiarity with TD, more than half of the participants still do not track TD (approximately 65%) and almost one out of five do it in an ad-hoc way (19%), i.e. by using tools that are not made for TD tracking and therefore are not effective. Finally, only 7% of the participants tracks TD in a more dedicated way.

An interesting observation is that the results are not significantly affected by the background and the role of the respondents. This datum increases the reliability of the results: independently of the organization and the background of the participants, we found very similar results across the respondents, which can be considered also more general. In other words, the means and the variance across different practitioners are similar in different organizations.

However, this also lead us to consider the following: different roles with different priorities and views (e.g. managers and developers) agreed on the estimated amount of effort done to keep TD at bay, as well as on the fact that such effort is not systematic (TD is mostly not tracked). Then, a unanswered question is: if TD is so painful, why do organizations not track TD in a more systematic way? One possible answer is that employees do not know how to track TD in an effective manner. This is supported by the fact that, most of the ones who track TD, do not use proper tools or documentation, while the few ones who systematically track TD are still doing it manually and with the rare usage of basic measurements. For this reason, we found important to propose the SAMTTD model, to help practitioners understanding what it means to track TD and what is necessary to implement a tracking process in practice.

Another answer to the current lack of TD tracking, despite the management effort, might be found in the results related to RQ8 and RQ9 concerning the necessity of a *Preparation* phase and its cost, which is critical for the introduction of a TD tracking process in the companies. The initial initiative needs to be conducted by one or more *champions* in the organization. An initial budget should be allocated, in order to allow the first activities

المنارة للاستشارات

www.m

related to the TD inventory, and this entails that there is a need for the commitment of management, which is achieved by communicating how a systematic TD management process would bring benefits to the organization. Unfortunately, this is one of the challenges reported by the practitioners, who claim that there is a lack of good instruments and publicly available results to advocate for the need of a systematic TD management. Other activities include the communication and alignment of what should be collected as TD, the set-up of measurement systems, the appointment of a Sub-System TD Responsible (SSTR) and the breakdown and distribution of the TD items to the teams. Unfortunately, the first investment can be burdensome, as a trial of 150 initial hours for a unit with three teams was barely enough to firstly identify the initial TD list, and did not leave time for the company to set up measurement systems and to accurately estimate and prioritize the TD items, although updating the TD backlog becomes lightweight in the following iterations.

For what concerns tools to track TD, we found that many participants use backlogs, implemented in project management tools such as Jira and Hansoft, and static analyzers. The results also suggest that these approaches require less management effort and they seem to give slightly more awareness about the TD in the system. However, it seems that, for most of the respondents, the awareness of the amount of TD present in the system is not affected by the tool in use, if not slightly. This means that TD tools are not only used by the teams to be aware of the TD, but rather for communication, monitoring and management purposes. The usefulness of these tools is shown by the fact that the participants using backlogs and static analyzers, spent less than the average (18-19% of the time compared to 25.9%) on TD management. However, the tools do not seem to help raising the awareness of the respondents, as the mean awareness remains between "somewhat disagree" and "somewhat agree". Many qualitative answers, both from the survey and from the case-study, also report the fact that many TD items cannot be automatically revealed, as they are too context-specific and they cannot be represented by generic patterns. This leads to the conclusion that better and more specific tools for managing TD need to be developed.

In summary, managing TD requires a few investments that are not well known by the practitioners and are difficult to be motivated by a precise cost/benefits ratio. Consequently, without an investment on processes and tools to track TD, it is difficult to make TD visible, as well as to advocate for refactoring "invisible" TD. This represents a *vicious cycle*: companies suffer the negative effects of TD and try to contain it, but at the same time they do not find enough motivations to invest in a more systematic management process. By looking at the motivations for start tracking TD, the results show that organizations do so when they experience the *interest* of TD: *slow feature development*, *quality issues* and *performance degradation*. However, at such point, the interest associated with TD is already high and, as explained in other recent papers [110], [23] from the authors of this manuscript, it is hard to refactor, as the cost has also increased and has become too expensive. In conclusion, the only way out the vicious cycle seems to be, for the practitioners, to *proactively start tracking TD*. Using backlogs and static analyzers help reducing the management overhead and increasing (even if slightly) the awareness of TD. New tools need to be developed, in two main directions: allowing the developers to communicate the urgency of refactoring TD to the

المنارة للاستشارات
www.r

management, and better (semi-)automatic tools to identify and track TD to increase the awareness of the respondents.

15.4.2. Related Work

There are two survey-based studies regarding the familiarity and tool usage related to TD. In [21] the authors concluded that 50% of respondents said that no tools were used, and only 16% stated that tools gave enough details. Their study also shows that 27% of the respondents do not identify TD. Furthermore, Holvitie et al. [22] show that over 20% of the respondents (in Finland) indicated poor or no TD knowledge. However, in these studies we cannot find a quantification of the estimated effort spent on TD management and there is no explanation about how a TD tracking process can be started or implemented. As a comparison with these studies, the results from our survey show that the familiarity of TD and its tracking seems to be higher among the respondents that answered our survey. Maybe this is related to the different size, culture or domain of the organizations, but given that our study is more recent, we could speculate that the familiarity of TD is growing. In our results, only 8.4% of our respondents was not familiar with TD, and 27% of the respondents used tools. Both findings are higher than in the other surveys.

There are a few articles about industrial practices concerning Technical Debt, for example [219], [149], [224] but they are single case-studies and, in two cases, they were performed in small companies. Also, such work does not focus on the current state of practice o technical debt tracking, a quantification of TD management effort, the motivations for starting tracking TD or the maturity evolution of tracking: this makes it difficult to compare the results with our survey, but we will take the topics one by one and discuss similarities and differences. As for the cost of tracking TD, [52] reports detailed results from a single case-study. Some results are in line with the broad results reported here: the effort might vary greatly, reaching even 70% of the development time, and starting the TD tracking is more expensive in the beginning but it becomes more lightweight when the process is repeated. In [225], several companies have been analyzed on their TD management process, which reports similar results to our cases, e.g. the limited use of measurements and lack of a systematic process. However, the study does not focus on TD tracking, it reports a broad snapshot of current practices and does not take change management perspective into account, in contrast with our work. For example, we report here information such as the quantified cost of managing TD, the reasons why organizations start tracking TD and the preparation activities and costs necessary to track TD. We present a maturity model, SAMTTD, that, taking into account change management aspects, allows for the transfer of knowledge to practice. This is visible by the additional four levels added in our model. We can consider the 4th step, in our model, as an especially important addition in our work, as we found evidence of a systematic process using TD-specific documentation, not reported in [225]. In addition, none of these studies report quantitative answers from as many as 226 practitioners, which also show trends and statistical results reported here.

There are a few studies regarding Technical Debt tracking and tools in literature. As for tools, most of the recent findings report tools created by researchers (e.g. [116], [226],

المنارة للاستشارات

www.r

[227]): the experience reports are usually related to the evaluation of the tool in a specific context, and therefore cannot be considered as state of practice (at least, not yet). This is understandable, as new tools are being developed while this manuscript is being written, and given that the attention on the TD topic, from software organizations, is quite recent. As for tracking, three initiatives have been reported in literature [226], [228], [229]: the first one[226], presents a tool called DebtFlag, which allows tracking TD and its propagation. However, the evaluation in practice of such tool has not been reported yet. The second one [228], reports the evaluation of a tool (AnaConDebt) to assess and track TD. A first study has been done in an industrial environment, but more studies are needed to understand if the tool is usable in practice. Finally, the last paper [229] reports a new method to analyze the TD reported in code comments. Although some of the features of the semi-automatic approach seem interesting, it is not clear how many TD items are covered by comments, and if this approach can actually be used in practice (the paper does not report a practical use of the method with an evaluation from the practitioners). For example, if we look at the survey conducted in this paper, currently only around 1% of the participants (three) state that they track TD using comments.

15.4.3. Limitations and Threats to Validity

Here we report the main threats to validity regarding this study, according to [73]: *construct validity, internal validity, external validity* and *reliability*.

Construct validity is concerned with the investigation device and the validity of the data with respect to the RQs that are investigated. In a survey, this is usually one of the main threats to the validity of the results, as participants might interpret definitions and other terms differently from each other. Although this phenomenon is unavoidable, we took a few approaches to mitigate the consequences. As for the misunderstandings related to the interpretation of what TD is, we have reported, before the questions, short definitions of the issues and management activities that are associated with TD according to the most up-to-date literature. In other words, we did not ask questions on "Technical Debt" directly, but on more concrete issues that are associated with it. In our experience, this should have reduced the possibility that the respondents would consider TD as something else, for example bugs or missing features (something that might happen in practice, according to our experience). We have also provided, in the last part of the survey, a definition operationalized from the various existing, formal definitions. We have asked a question about if the practitioners were familiar with TD according to the definition, and mostly they agreed. Although this does not ensure that the practitioners had answered with a full knowledge of what Technical Debt is, we believe that the two mitigation strategies together had contributed to reduce the threats to construct validity.

There is a threat of internal validity when mapping the respondents to the levels in the SAMTTD, as we did not ask this question directly to the participant. To mitigate this threat, we used multiple evidences from various quantitative and qualitative answers, and we can reliably say that no company is using integrated measurements of TD, which place the respondents necessarily from level 1 to 4. We have thoroughly assessed the number of respondents for level 3 regarding the usage of a tracking tool. By definition, the respondents in level 1 do not know what TD is, and this datum comes directly from the

المنارة للاستشارات
www.m

answers related to their familiarity with TD. Level 4 include the few practitioners who have confidently reported how they track TD. These practitioners have also been interviewed, which yielded a description of what systematic process they were used. Consequently, level 2 contains the remaining of the respondents not included in level 1,3 and 4.

As for the results concerning testing hypotheses statistically, it is important to notice that, in most cases, we could not reject the null hypotheses that the results would depend to the background of the respondents (roles, company, etc.). This means that we could not find enough evidence, in this dataset, to support the rejection of the null hypotheses, but the reader should be warned that we also did not prove the opposite hypotheses.

Finally, it is important to report the threats to external validity: we investigated mostly large, embedded system companies and from the Scandinavian area. This entails three possible threats:

- It is possible that, in other domains (e.g. web development), the % of the companies in the maturity steps would differ. To mitigate this threat, we have included a company developing "pure" optimization software. In this case, we did not find a statistical difference with respect to the other companies. However, more research is needed to understand if there is a difference.
- Companies in other countries, with different contexts and cultural background, might answer the survey in a different way or have different ways of managing Technical Debt.
- Small companies might behave very differently with respect to Technical Debt management.

Therefore, the reader must be aware that there are some limitations to the extent to which we can generalize from these results.

There are also threats to the reliability of the results, or else, the results might be biased depending on a particular interpretation given by the authors, method, or particular source of evidence (e.g., if we asked only developers but not managers), as reported below:

- There is a threat in the quantities estimated by the respondents with respect to Q1. We do not know what the given estimations are based on, since most of the participants do not track TD and the time spent on it. However, as the demographic data show, many participants can count several years (more than 10) of software development experience. This means that they are used to estimate the amount of work that has been done or that is upcoming, which mitigates the threat that the estimated effort would be very distant from the real one.
- As for the authors' interpretation, we have made sure that, especially for the qualitative data analysis, we have applied observer triangulation: two or more authors have analyzed the interviews and either separately coded the statements, or checked the other authors' codes. Although this does not remove the threat completely, it is the main strategy used when qualitative data analysis is involved in the study.

المنارة للاستشارات
www.r

- Relying only on quantitative data might miss important details that are necessary to understand the results or might show correlations that are not related to any real causality. For example, we could not find reasons, from the quantitative background data, that would explain the variance in the amount of time that the participants are employing to manage TD. However, we could combine the quantitative results to qualitative answers coming from some of the organizations participating in the survey, which helped explaining the factors related to their maturity by analyzing the interviews.
- Finally, there is a threat of reliability of the results, as the percentage of developers participating in the survey was larger than other roles. This mean that the results might be skewed by the developers' biases. However, to mitigate this threat, we performed a chi-square test to understand if the distribution of the answers would depend on the roles of the respondents. The test did not support such hypothesis, meaning that there was not a statistically significant difference between different responding roles (different roles gave similar answers). By having such roles participating in the survey, we could apply a mitigation strategy denoted as source triangulation.

15.5. Conclusion

According to 226 respondents in 15 software organizations, practitioners estimate to spend, on average, a substantial amount of time trying to manage TD (25%), although such amount is affected by some variance. Software companies in Scandinavia are more familiar with the TD metaphor with respect to previous studies, and they track TD more. The awareness of TD in the system seems to be somewhat known by the developers, independently from which approach is used. Tools such as backlogs (the most popular approach) and static analyzers help reducing the management overhead of approximately 7%. However, only 26% of the respondents use tools to track TD, and only 7.2% of them created a systematic process in their organization. This is due to the lack of knowledge of what is necessary to implement to introduce a TD tracking approach in the organization, in terms of tools and processes, as well as a lack of awareness of what the negative effects of TD are before they occur. Moreover, we studied some approaches and found that an initial investment on *preparing* for the introduction of TD is necessary, which makes starting TD tracking less appealing for managers who need to fund the activities. However, although there are some obstacles to overcome, some of the companies are proactively and strategically implementing a solution to make TD visible, which shows that it is practical to introduce such approaches. To help this process for other practitioners, we propose a Strategic Adoption Model (SAMTTD), based both on the evidence collected across this study in combination with current literature. The Model can be used by practitioners to assess their Technical Debt tracking process and to plan the next steps to improve their organization.

المنارة للاستشارات

www.r

www.m

16. Technical Debt, from a Startup Company Perspective

In this chapter, we seek to understand the organizational factors that lead to and the benefits and challenges associated with the intentional accumulation of technical debt in software startups.

Software startups are typically under extreme pressure to get to market quickly with limited resources. This pressure is likely to cause startups to accumulate technical debt as they make decisions that are more focused on the short-term than the long-term health of the codebase. However, most research on technical debt has been focused on more mature software teams, who may have less pressure and, therefore, reason about technical debt very differently than software startups. In this study, we interviewed 16 professionals involved in seven different software startups. We find that the startup phase, the experience of the developers, software knowledge of the founders, and level of employee growth are some of the organizational factors that influence the intentional accumulation of technical debt. In addition, we find the software startups are typically driven to achieve a "good enough level," and this guides the amount of technical debt that they intentionally accumulate to balance the benefits of speed to market and reduced resources with the challenges of later addressing technical debt.

This chapter has been published as:
Embracing Technical Debt, from a Startup Company Perspective
T. Besker, A. Martini, R. E. Lokuge, K. Blincoe, and J. Bosch
IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 415-425, 23-29 Sept. 2018.

16.1. Introduction

Software startups are freshly created companies with no operating history and mainly oriented towards developing high-tech and innovative products, aiming to grow their business in highly scalable markets [60], [6]. Compared to more mature companies who are often maintaining software to an established market, software startups face different types of challenges. Startups often operate with limited resources and under extreme time pressure as they strive to produce their product and avoid being beaten to market by a competitor or running out of capital [3]. Thus, startups typically develop early software versions to test and validate emerging ideas to avoid wasteful implementation of complicated software which may be unsuccessful in the markets [230]. Under these conditions, often the extra effort required to design and implement software with an optimal design is considered an unaffordable luxury and a potential waste of time and effort.

Software companies often make sub-optimal design decisions to allow them to get to market quickly [3]. For instance, the product might be built with an inflexible architecture that cannot be easily changed to speed up time-to-market and let the startup put their

المنارة للاستشارات
www.r

product in users' hands earlier, get feedback, and evolve it [111]. If and when the developed software becomes successful on the market, then the pressure turns modifying the software to meet the user needs (i.e., adding new features). This can cause startups to build upon the original inflexible architecture that was not designed to last for the long term and is not easily extendable.

The result of this situation is the accrual of what is described as Technical Debt (TD). The TD metaphor was first coined at OOPSLA '92 by Ward Cunningham [8], to describe the need to recognize the potential long-term negative effects of immature code that is made during the software development lifecycle. A recent, definition was provided by Avgeriou et al. [4] who define TD as "In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability".

TD has been the focus of much recent research, but this research has been mostly focused on mature software companies, where large amount of TD is considered to be detrimental to the long-term success of software development [7]. However, deliberately accumulating TD could be much more beneficial since it can considerably speed up time-to-market, allowing them to release their product to end-users faster, get feedback, evolve the software, and preserve capital [231]. However, TD must be managed to ensure it is addressed at an appropriate time; unmanaged TD can have negative consequences, such as the death of the startup itself [232].

There is a current paucity of empirical research focusing specifically on TD and startups [233]. This paper reports on a qualitative study that examines the organizational factors that influence the introduction of TD and the benefits and challenges of deliberate taking on TD. Through interviews with 16 professionals at seven different startups, we identified six organization factors that lead to TD. In addition, we present a list of benefits and challenges of TD in startups, which can be considered by practitioners to aid them in the TD decisions.

The remainder of this paper is structured as follows: In Section II we describe the background and related work. Our research methods are described in Section III. We describe the cases in Section IV. The results are presented in Section V. Finally, we discuss the implications and limitations of our work in Section VI, and offer a brief conclusion in Section VII.

16.2. Background and related work

In this section, we provide a complete description of a software startup, provide some background on the startup lifecycle, and review related work on TD in startups.

المنارة للاستشارات

www.r

16.2.1. Software Startups: A Definition

Giardino et al. [6] define software startups as those "organizations focused on the creation of high-tech and innovative products, with little or no operating history, aiming to aggressively grow their business in highly scalable markets". Sutton [234] presents different characteristics that reflect both engineering and business concerns, which software startup companies must operate within. Software startups are relatively young and inexperienced compared to more established and mature development organizations, and they commonly have very little accumulated experience or history. Typically, their resources are limited, and they primarily focus on getting the product out, promoting the product, and building up strategic alliances. Their business is dependent on influences from various sources, such as investors, customers, partners, and competitors. The software these startup companies are developing are commonly technologically innovative products, and their developing often involves cutting-edge development tools and techniques [234].

16.2.2. Software Startups Life Cycle

Crowne [235] identified four distinct stages for a software startup: startup, stabilization, growth, and maturity. Each stage has different types of critical product development issues that potentially can lead to company failure. The first "Startup" phase refers to the period between product idea and the first sale. This stage is characterized by a product where the product doesn't meet the customer's requirements and is unreliable and fails frequently. Rectifying defects takes longer than expected and often creates additional defects [235]. The second "stabilization" phase begins when the first customer takes delivery of the product and ends when the product is stable enough to be commissioned without any overhead on product development. During this stage, a divide between developers can be spotted, where the developers who join the company early, and those who are recruited later differ in terms of that the early developers mount significant resistance to organizational change. During this stage, the non-functional requirements such as security, reliability, scalability, and performance gain additional attention, and the result of the previously introduced sub-optimal solutions becomes evident [235]. The third "growth" phase takes place when the product can be commissioned for new customers without creating any overhead on the development team. This phase ends when market size, share, and growth rate have been established, and all business processes necessary to support product development and sales are in place. In this stage, new features implementation requires a coordinated program of activities across functional areas including product development, professional services, support, and sales and marketing, which stresses the importance of having a repeatable process for software development implementation. The last "maturity" stage occurs when the company has evolved from a startup into a mature organization, where, e.g., market size, share, and growth rate have been established In this stage also all processes necessary to support product development and sales are in place [235].

المنارة للاستشارات

www.r

16.2.3. Startups and Technical debt

There is a lack of research studies on TD management in software startups [233]. Giardino et al. [6], conducted an empirical study addressing how startups employ software development strategies, using a Greenfield Startup Model (GSM), which also covers startups and TD to some extent. Giardino et al. describe that to be faster, startups may introduce TD as an investment, whose repayment may never come due, with the long-term negative effects on morale, productivity, and product quality. Further, in their study they state that "Startups achieve high development speed by radically ignoring aspects related to documentation, structures, and processes", and that "instead of traditional requirement engineering activities, startups make use of informal specification of functionalities through ticket-based tools to manage low-precision lists of features to implement, written in the form of self-explanatory user stories".

Gralha et al. [236] investigated the evolution of requirements practices of software startups. They found that TD is one of the six factors that influence the requirements practices of a startup. They identified three phases regarding the accumulation of TD in startups. They also identified trigger points that cause startups to transition from one phase to the next. An increase in the number of employees and software features causes startups to transition from simply knowing and accepting TD to tracking and recording it. Then, when their client retention rate goes down, or they begin to see an increase in negative feedback, they begin to manage and control TD.

Another study which to some extent covers TD in startups is presented by Yli-Huumo et al. [237]. In that study, they investigate the relationship between business model experimentation and TD, with the goal of understanding if conducting these types of experimentations have any effect on the amount of TD occurring during the software life cycle. The concept of a business model experimentation in their study refers to when a company uses the technique to validate assumptions made on a product from real customers before the actual product is created. An example of this can be illustrated when a Minimum Viable Product (MVP) is used to test the business model by collecting and measuring customer feedback [237]. Since adopting the technique of business model experimentation is a conventional approach in both startups and larger companies [237], this study is somewhat related to ours. The result of their research showed that there is a relationship between business model experimentation and the occurrence of TD and also that focusing too much on business model experimentation and not on remediation of TD can have consequences to the product quality.

In a research agenda for software startups provided by Unterkalmsteiner et al. [233], the authors state that researchers must build a more comprehensive, empirical knowledge base to support forthcoming software startups. They list several research question related to TD, and by answering these questions, they state that it could help clarify the role of design decisions in software development in the context of a software product roadmap, similarly to what happens in other engineering disciplines. The overall goal of the research questions listed by Unterkalmsteiner et al. [233] address in what way practitioners will be able to make better decisions considering the characteristics of the current software product implementation.

المنارة للاستشارات

www.r

16.3. Research Methodology

The goal of this study is to understand how software startups reason about TD. In particular, we are interested in the organizational factors that impact TD together with the potential benefits and challenges of TD. We, therefore, aim at answering the following research questions:

RQ1: What organizational factors influence the accumulation of TD in software startups?

RQ2: What are the challenges and benefits of Technical Debt for software startups?

In order to answer these research questions, we investigated the strategy of software development in different software startup companies by interviewing 16 practitioners in seven different startup companies, working in seven different areas.

16.3.1. Participants

We collected data from software professionals active in seven different software startup companies, shown in TABLE I. The sample population was selected using a non-probability sampling technique [86], where the selection of participant companies was obtained using convenience sampling. The startup companies were located in two different countries. The companies are described in more detail in Section IV.

TABLE I - STUDY PARTICIPANTS

Role	Company	Country	Segment
Developer	A	Country 1	Sport
Developer			
Developer	B	Country 1	Energy
Developer			
Developer	C	Country 2	Retail
CEO / Developer			
Co-founder / Developer			
Co-founder / Developer			
Co-founder / Developer			
CEO	D	Country 2	Medical
CFO			
COO			
Senior architect			
Advisor (Business and Technology)	E	Country 1	Media

المنارة للاستشارات

www.r

Role	Company	Country	Segment
Advisor (Business and Technology)	F	Country 1	Software Development
Chairman of the board	G	Country 1	Mental Health

16.3.2. Data Collection

Initially, we ran two workshops (one in each country) with participants from four different startups (A, B, C, and D). The workshops included both a presentation made by one of the authors about TD, followed by a group discussion where the participants explored their own experiences with TD within their startup companies. Each workshop lasted about 120 minutes and in total 12 practitioners from the investigated startup companies participated.

The goal of these workshops was to introduce the participants to the study, to align and equip them with relevant knowledge about the concept of TD and to gather background and contextual information on each participating startup company in preparation for the following interviews.

We conducted semi-structured (as suggested in [73]), face-to-face interviews with 16 professionals from seven different companies, from two different countries. To improve the reliability of collected data at least two of the authors participated in each interview session. Each interview lasted between 60 and 120 minutes and was digitally recorded and transcribed verbatim. The questions were prepared by three of the authors together.

The aim of the interviews was to understand the accumulation and refactoring of TD and what contextual aspects (related to the startup's environment) influenced such accumulation. We started by asking participants to describe their startup company and product. We asked follow-ups to learn about the contextual aspects of the startups, focusing on the contextual characteristics described in [60]. In the remainder of the interview, our questions focused on TD. Specifically, we asked:

- Describe some critical TD issues.
- Which TD issues were refactored (and when)?
- Which TD issues are planned to be refactored (and when)?
- If TD issues are not planned to be refactored, why not?
- What value did the accumulated TD give the company?
- What cost was (or will be) paid to remove the TD?
- What extra costs were (or will be) paid because of the TD?
- What led to the accumulation of TD?
- What roles, processes, guidelines, and strategies were used for TD?

المنارة للاستشارات

www.r

Finally, to get more insight into the existing TD, we also jointly ran the software SonarQube [238], and AnaConDebt [239] during the interviews. None of the companies previously used these tools, and they were not familiar with the output from the tools in advance. We asked questions on:

- What issues were revealed and were they already known?
- Would it have helped to use the tool (and when)?
- Will you use the tool in the next iterations?

16.3.3. Data analysis

We used thematic analysis [84] to identify, analyze, and report patterns and themes within the interview data. Thematic analysis involves searching across a dataset to find repeated patterns of meaning. The thematic analysis provides a flexible and useful research tool, which offers a detailed, and yet complex account of the collected data.

The thematic analysis was conducted using a six-phase guide. First, the audio-recorded qualitative data collected from interviews were transcribed, and we familiarized ourselves with the data through careful reading of the transcripts. The second step involved the production of initial codes from the data, where we organized the data into meaningful groups. In this phase of the analysis, a Qualitative Data Analysis (QDA) software package called Atlas.ti was used. The third phase focused on searching for themes by sorting the different codes into potential themes and collating all the relevant coded data extracts within each identified theme. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the citation "*if it [the software from a third-party application] lifts and take off, we can build our own solution*" was coded as "*Third party*" in the theme "*Software development Process*." To ensure that the coding was performed in a consistent and reliable fashion and in order to triangulate the interpretation of the data and to avoid bias as much as possible, two authors synchronized some of the output of the coding, following guidelines provided by Campbell et al. [85]. The fourth phase focused on the revised set of candidate themes, involving the refinement of those themes. The refinement concentrated on forming coherent patterns within the themes. When needed, we revised the themes or created a new theme. The fifth phase focused on identifying the essence of each theme and determining what aspect of the data is captured by each theme. This phase also stressed the importance of not just paraphrasing the content of the data extracts, but also identifying what is interesting about them and why.

The final phase of the thematic analysis took place when we had a set of fully developed themes, and involved the final analysis and write-up of the publication. We have made a figure illustrating how the codes and the corresponding themes were assigned during the thematic analysis available at https://figshare.com/articles/Thematical_Analysis/-6115172.

المنارة للاستشارات

www.m

16.4. Description of cases

In this Section, to provide more context for our study, we describe the companies in more detail. We also indicate the startup stage for each company (using the stages in Crown's [235] classification of startups, which we described in Section II.B). Across the seven cases, all stages are represented by at least one of the cases in this study.

In this Section, to provide more context for our study, we describe the companies in more detail. TABLE II. summarizes the seven companies that participated in this study. As can be seen, there is diversity across all aspects. We also indicate the startup stage for each company (using the stages in Crown's classification of startups, which we described in Section II.B). Across the seven cases, all stages are represented by at least one of the cases in this study. Figure 1 shows how TD was accumulated or addressed in each stage. All companies reported accumulating significant TD in the startup phase. Surprisingly, two companies reported undertaking either a major refactoring or a complete redesign during the startup phase prior to securing their first customer. Both of these cases were due to unintentional issues with the code or the design. During the stabilization phase, most companies reported addressing the TD that accumulated in the previous stage either by taking on formal refactoring initiatives or by informally removing TD as needed. The two companies in the growth and maturity stages indicated that most of the TD had been addressed before entering these stages. Only two of the companies, C and F, had not yet performed a large refactoring or redesign, but both planned this for the future.

TABLE II - DESCRIPTION OF CASES

Company	Product	Domain	Years since founding	Founders SW Knowledge	Software developed	Current Employees	Experience of Software Developers	Development Practices
A	Mobile app	Sport	2.5	None	Initially external then in-house	Founder, CTO, CMO, 3 developers, one salesperson	2 junior, 1 senior + senior CTO	Some agile practices (e.g. sprint planning)
B	Mobile and web apps	Energy	6	High	In-house	CEO, 5 developers, two sale reps	4 senior, 1 junior	Scrum
C	Web app	Retail	2	High	In-house	4 Founders	All junior	No formal process
D	Web app	Medical	2	None	In-house	3 Founders, 2 Technical staff	All senior	Some agile practices (e.g. Kanban, CI)
E	SaaS app	Media	9*	Low	In-house	35 employees (Two-thirds are developers)	All junior	Some agile practices
F	Web app	Software	2	High	Combination in-house and consultant	Founder + consultant as needed	Senior	Scrum
G	Mobile app	Mental Health	6	None	Initially external then in-house	Founder, CTO, 3 developers, 1 salesperson	3 junior + senior CTO	Scrum

* Today this startup is 9 years old, but the data collected for this startup reflects a time period of 3-5 years after they were founded

المنارة للاستشارات
www.n

Startup F	Startup A	Startup C	Startup D	Startup G	Startup B	Startup E	
Beta version of product out to gather feedback; Intentionally introducing significant TD to release quickly	Accumulated large amounts of TD	Accumulated large amounts of TD	Accumulated large amounts of TD; Major refactoring due to unintentional TD by junior developer (later replaced by senior dev)	Accumulated large amounts of TD; Complete redesign/rebuild due to unintentional TD from sub-optimal architecture and design decisions.	Accumulated large amounts of TD	Accumulated large amounts of TD	Startup
Plan: address feedback; release commercial version; large refactor	Curerently undergoing refactoring initiatives; Adding new features to product.	Removing TD on as-needed basis; No current plans for major refactor.	Removing TD on as-needed basis; No current plans for major refactor.	Scalability and usability improved from previous refactoring; Removing TD on as-needed basis.	Several refactoring initiatives to reduce TD.	Customer-specific product versions causing sub-optimal solutions and accumulation of TD; Switched to generic product version to improve scalability.	Stabilization
		Plan: rewrite full code base to increase scalability			Very little TD, actively managing TD.	Introduction of new products required a coordinated program of activities across functions (e.g., development, professional services, support, sales and marketing).	Growth
						Generic versions of all products, minimal TD.	Maturity

Figure 1. Overview of TD strategies across Crowne's [11] stages for each Startup

16.5. Results

The following subsections present results for the research questions presented in Section III, and the results are grouped according to each research question.

16.5.1. What organizational factors influence the accumulation of TD in software startups? (RQ1)

Our analysis has identified many factors that influenced the amount of TD that the startups accumulated.

16.5.1.1. Experience of software developers

Our results indicate that the experience level of the software developers can have both positive and negative influence on the accumulation of TD. As startups are initially very small in terms of number of developers, the experience level can be much more impactful on startups than on more mature software teams.

Less experienced (junior) developers often unintentionally accumulate TD due to their lack of experience. As one interviewee from Company A stated, "It's really good to have at least one guy that is more experience in the team." Another interviewee from Company E explained this as: "Junior developer are less able to project outcome to the future about how the system is likely to evolve, which means that they have a tendency to focus on the 'here and now', and solve the today's requirement whereas people that are experienced can often predict a little bit more easily what is likely to come in the future and already

المنارة للاستشارات

www.m

start to prepare the system for that…Finding the right balance between focusing on now and prepare for the future is a very different balancing act for startups then it is for large scale companies." Thus, junior developers are more likely to introduce unintentional TD due to their lack of experience.

More experienced (senior) software developers are more aware of and have accumulated more experience about the effect of introducing TD, compared to junior developers. Thus, having senior developers to guide the development is very beneficial. However, senior developers are more expensive, and startups typically cannot afford to have many senior developers. "I think that it would be very expensive to get another very experienced person. And maybe it's not worth it."

In addition to high salary costs, senior developers may be less likely to intentionally accumulate TD if they have experience working on more mature software products that are not under such extreme time pressures to get to market. A participant from Company D stated, "If we had had the knowledge or the insight, we probably would have taken on board technical debt earlier on, but I think because we ended up hiring senior developers that were used to, working in certain ways with testing and re-testing everything. They ended up building, a fairly robust, as far as we can tell, but for our purposes, there might have been something over-engineered perhaps." Senior developers may be less willing to operate in an unstructured and less quality oriented approach. For example, one interviewee from Company A said: "So, you need to be more flexible, and if you are senior maybe you don't are ready to cope with that." This could cause startups delays in getting to market if TD is always avoided in favor of producing high quality software.

16.5.1.2. Software knowledge of startup founders

We found that the knowledge of the founders, related to software development, has an impact on how TD is accumulated. Founders with limited software development knowledge are less likely to accumulate TD intentionally. Since they are unable to implement the product themselves, they are likely to employ an external consultancy company or hire in-house developers to implement the first software solution, which involves a significant investment prior to being able to receive revenue from the software. The founders typically expect a high-quality implementation in return for this investment since they tend to have no knowledge about the benefits of TD.

On the other hand, when the startup founders are experienced software developers, they are more likely to implement the product on their own. They often accumulate a large amount of TD because they focus on producing the first release quickly. They view the initial release as more expendable since they have not invested money towards its development.

16.5.1.3. Employee growth

We found that when startup teams were remaining stable in terms of the number of developers, they did not feel a need to reduce their TD since the issues related to the TD affected only the developers, not the customers. The participants did not believe their TD

المنارة للاستشارات

www.r

impacted product performance or usability. While the TD did make the code more difficult to extend or modify, the existing developers were already familiar with the TD in the code, so it was not necessary to reduce the TD.

However, we found that the addition of new developers caused the TD to decrease for several reasons. First, the existing developers reduce the technical debt prior to hiring new developers. The developers want the code to be easier to understand so that new developers can be onboarded more quickly. They also do not want new developers to unintentionally introduce additional technical debt because they are modeling their own code on existing TD. For example, an interviewee at company B stated: "But as time goes on, the quality of real code, or its readability and how easy it is to work with, becomes more and more important. It is very easy when you as a developer comes into a project that you start writing code in the way of the existing code base. You kind of go 'oh, this is how they do it here,' and that is not always a positive thing. A lot of time that is quite a negative thing, because, you slip into those habits and before you know it, all the things that you personally hold true about what good code is, you are not doing that anymore". This fear of that new employees will directly copy the code, and thereby duplicating TD was also described by one interviewee from Company A stating: "And if you come in as a new developer, you might copy-paste some code, and you copy-paste that old thing of doing it, and we get the more messy code. And that is what we don't want."

In addition to the existing developers purposely reducing TD, new developers also remove TD as it is difficult to extend. The existing developers may be so familiar with the code, that they no longer notice the problems, while they will be more obvious to the new developers. For example, a developer from Company D said "I mean there's a big refactor when they brought me on. …[we] ended up throwing a lot of code out and rewriting it. And that was probably because of the technical debt side of things in there, using constants throughout and the like."

16.5.1.4. Uncertainty

In general, uncertainty about the future of the organization and product is very common characteristic in the startup companies. Our results suggest that, not surprisingly, the uncertainty plays a major role when making decisions about TD. One of the interviewees from Company C put this as "with these sorts of projects, you need to build a business case, and you'd be silly to like build something with no technical debt in it until you've at least proven that it's something you have to pay for. As soon as we confirm that there will be [revenue], and see the money starting to come in, that's when you probably start to look at the repaying the technical debt". Another participant from Company D stated "there was a point where basically we said, okay, now we just need to stop spending money because we don't know if this is even going to be a viable project and if it's going to generate any money or anybody's going to want to buy it".

This uncertainty causes startups to accumulate significant TD so they can release a proof-of-concept as quickly as possible. Once their idea is validated and they have a number of paying clients, they can worry about paying off their TD – possibly be rewriting the entire codebase from scratch.

المنارة للاستشارات

www.r

16.5.1.5. Lack of development process

None of the interviewed startup companies adopted a systematic software development process, and the need of having such a process was not considered by the interviewees to be important during the first phases in the startups' life-cycle. However, this topic was brought up as a challenge, especially when the startup grows and hires more developers. A lack of processes for the management, identification, and prioritization of TD means that TD decisions are often made ad hoc, and there are no consistent decisions being made across the team. This is especially important as the team grows to ensure there is conformity. As one interviewee in company A said: "Multiple ways of doing things, are spreading at the same time… I mean, it is quite important for me, when we start to grow, that we have the same way of writing code."

16.5.1.6. Autonomy of developers (related to TD)

Related to the lack of development process, developers often have full autonomy to decide when to take on TD and plan when to refactor the TD. Developers typically do not discuss TD-related decisions with others. While this allows for flexible work and short decision paths, it means developers, who are often not financially invested in the project, are making very important decisions without possibly considering the financial repercussions of these decisions.

This can be especially problematic when employing external software consultancies since decisions tend to made based on the benefits to the consultancy company, rather than making the best decision for the software product under development. The consultancy could decide to minimize TD because they want to maintain a high-quality reputation for their company and do not want to deliver software that is not maintainable. If the development is not on a fixed price contract, this desire for perfection could cost the startup significant time and money. On the other hand, they may be driven to take on significant TD since they know they do not need to maintain the software and they are driven by the desire to save money during the development. For example, the interviewee from Company G stated: "the externally hired consultants, they just did what was asked of them in their contract, with the lowest possible development effort. That is commonly how it works with externally hired developers, they do not really care about Technical Debt, they care about delivering the software according to the given specification they are paid for."

We saw only one case where developers were not given full autonomy regarding TD decisions. The founders of this company found being involved in even trivial implementation decisions very useful. One of the founders of Company D said "I think that they got used to basically involving us in their decision-making even though on a relatively trivial scale so that they'd ask about everything… And then we could understand and be involved in making those decisions about, how much debt and things will take on, even though we didn't call it debt. And there was a point probably about two-thirds of the way through the project where 'cause we'd often get updates on estimates of hours required to complete certain tasks so we'd keep an eye on how much money we were spending."

المنارة للاستشارات

www.r

TABLE III - ORGANIZATIONAL FACTORS INFLUENCING TD IN STARTUPS

Factor	Level	TD	Reason
Experience of developers	low (junior)	increases	poor design decisions (unintentional)
	high (senior)	increases	aware of benefits of TD (intentional)
		decreases	used to produce high quality software
Software knowledge of founders	low	decreases	unaware of benefits of TD, a large investment for developers causes desire for high-quality
	high	increases	develop product themselves, code viewed as expendable
Employee growth	stable	stable	devs already familiar with code (and its TD), no impact to customer
	increasing	decreases	existing devs refactor to make onboarding easier
			existing devs refactor to prevent a culture of “bad” code
			new devs refactor because code not readable
Uncertainty	high	increases	reduce development time and cost
	decreasing	decreases	TD repaid after market validation
Lack of development process	---	varies	ad hoc decisions
Autonomy of developers	high	varies	developers make decisions without any guidance (possible poor business decisions)
	low	varies	strategic decisions made

Answer to RQ1: We identified six organizational factors that influence the accumulation of technical debt: experience of developers, software knowledge of startup founders, employee growth, uncertainty, lack of development process, and the autonomy of developers regarding technical debt decisions. The results are summarized in TABLE II.

المنارة للاستشارات
www.r

16.5.2. What are the challenges and benefits of deliberately introducing Technical Debt for software startups? (RQ2

In this section, we explore how software startups determine and reason about both the challenges and benefits of intentionally introducing TD. In general, startup companies deliberately introducing TD, have a positive attitude of doing that. They are also relatively aware of the harmful effects these decisions can have on the future software in terms of impeding innovation and expansion of their software systems.

16.5.2.1. Benefits of intentional technical debt

We identified many benefits of intentionally introducing TD in software startups.

Cutting development time in order to be able to release the product as quickly as possible is seen as a large benefit for startups. Getting to market quickly can:

- **enable fast feedback** from the customers. An interviewee in Company A said: "We prefer to cut some corners to improve the speed, and get something out instead of making it more mature directly" …."It is more important to get to the market fast and get feedback from the users, then to focus on avoiding TD, taking on TD is ok."
- **increase revenue**. One of Company C's founders said, "Yeah, we probably wouldn't have got the contract earlier, right… Then we wouldn't have the capital". Another participant from Company A said "we are a startup, and we need to make money. We need to get things working, but they don't need to be perfect".

Another benefit is the **preservation of startup capital** since commonly startup companies have less money in the early stages. A participant from Company D stated, "it's just that we had to get the code out the door. And we had to get it so that we could afford it." Another participant from Company F said "by taking the first technical debt, we spent 10% of what we would have spent if we would have done the whole product without TD."

Related to saving money and time, another benefit is the **decreased risk**. Since the startups involve uncertainty, it is sometimes wise to invest as little money and time as possible prior to validating the idea through evaluation of the product. A participant from Company F said: "In case the product would turn out to be a failure, we would have saved 90% of the money…we avoided a big risk, and we reduced uncertainty thanks to technical debt. It was a great decision, I think."

Intentional TD also allows startups to stay flexible. When they do not spend large amounts of money or time developing new features, they are more willing to discard them and alter the product significantly when needed. Thus, the TD allows them to make more objective decisions. "If you put too much time and effort in there, it could be harder to

المنارة للاستشارات

www.r

throw it away in the next version. So, I think it's not always bad that you don't do the best”.

16.5.2.2. Challenges of intentional technical debt

Despite the benefits of intentional TD, we also identified challenges since the sub-optimal solutions would eventually need to be fixed.

The two investigated companies who initially hired an external consultancy company to implement the first software solution failed in doing so. Most of the initial implementation was later removed and replaced by in-house developers, causing significant delays and additional expenditures. In such extreme cases, TD can cause the **product failure** or a **business disruption**.

Another challenge of TD is the **reduced scalability** it often introduces. “If you validated it and it’s looking good, you wanna be able to put your foot on the gas and go quickly and scale. And if the architecture’s not ready…” A first, light and sub-optimal solution may only work in a specific setting but will need to be refactored in order to scale the software. One developer from Company A put it “growing is not just like taking what we have and do the exact same thing because that will only scale to a specific limit…There was no segmentation of the code in any part. We started to split the code up, we started to segment and to separate the code, so that we also can scale different part of the code.”

The interviewees mentioned different TD types such as architectural, infrastructural and source code related TD as having a substantial negative impact on the system growth.

Another challenge is that the harmful effects of TD increases in severity as the software grows and when more developers were involved in the development process. Thus, the introduction of TD can have **compounding effects** on the development time and resources, since it will take more time to develop code on top of existing TD. Then, if the TD is removed later, it all of the code built on top of the TD will also potentially be impacted. As one interviewee at Company B put it: “In a greenfield project, I think there is an argument hacking together something that works quickly. But as time goes on, the quality of real code, or its readability and then how easy it is to work with, it becomes more and more important.” Another challenge is that fixing TD could **increase risk**. When fixing TD, it might create new bugs in the code, adding to the amount of future work that needs to be done. “The bugs will probably grow, especially if we try and fix it, spend time trying to fix it.”

Finally, the introduction of TD requires the **loss of productivity to be managed later**. We found that during the early phases, startups rarely manage their TD and decisions are often made on an ad hoc basis. For the four companies who ran SonarQube during our study (A, B, C, and D), they were all surprised that more TD than expected was identified. All the startups felt such a tool would be useful to manage TD in the future. As the founder from Company D said, “I think this is very useful in terms of prioritizing the, you know, the back end of what we have and what we need to sort of like work on.”

المنارة للاستشارات

www.r

16.5.2.3. Good Enough Level

When startup companies deliberately introduce TD, they implicitly decide what a Good Enough Level (GEL) of the software quality is and what amount of TD is acceptable to take on. They weigh the benefits and challenges of the TD when making their decisions (illustrated in 1). However, it is not usually an easy decision. A founder of Company D said "It's difficult to balance where you're constantly making decisions how do we balance what we're spending on this, versus the likelihood of producing these results."

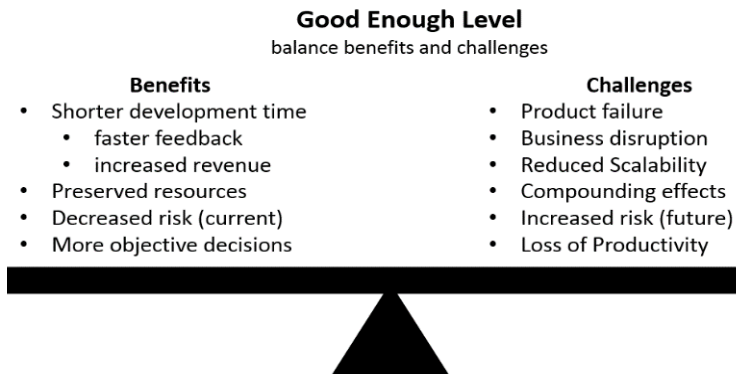


Figure 2. Good Enough Level is achieved by considering the ideal balance between the benefits and challenges associated with intentional TD.

Answer to RQ2: Intentionally introducing technical debt allows startups to cut development time, enabling faster feedback and increased revenue, preserve their resources, decrease risk, and make more objective decisions. However, the technical debt causes reduced scalability, becomes more severe as the product grows, and introduces future development risks. Thus, deliberately introducing technical debt brings both benefits and challenges and startups must weigh these to determine a "*Good Enough Level*".

16.6. Discussion

In this section, we discuss recommendations for startups, compare our results to existing knowledge on accumulation and refactoring of TD in other contexts, and describe the limitations of this study.

16.6.1. Recommendations for software startups

Based on the finding related to the organizational factors that influence TD in startups and the benefits and challenges associated with TD, we have the following recommendations for startups.

Balanced experience levels (of developers) needed. We found that junior developers often introduce unintentional TD. Senior developers are often more calculated in their TD

المنارة للاستشارات
www.r

decisions. However, senior developers may be less risk adverse, having more experience working on more structured, mature products where quality is paramount. A mix of both senior and junior developers seems ideal to find the right balance between TD and quality. These results are in line with the ideas of Crown [235], who states that "The principal developer for the company must be highly experienced, and familiar with all aspects of software engineering practice. This person must also be an accomplished technical leader, as they will need to influence their less experienced colleagues". Though, we advocate that junior developers are equally important.

Unbiased technical advisors needed. When the startup founders do not have software development knowledge, those implementing the software are likely to make decisions that benefit their own needs, rather than the startup company. For example, they may cut corners to save their own time, or they may gold plate the software to build up their own reputation (and to increase their own revenue). Thus, startup founders who lack software development expertise should consider seeking technical guidance from someone other than the company or developers they hire to implement the solution so they can obtain unbiased advice related to TD decisions. Depending on the stage of the startup (and the available capital), this advice could be obtained by the introduction of a CTO or from an external consultant.

Consider "contagiousness" of TD in prioritization. We found that TD is often removed as the number of developers increases. This is in line with the results of Gralha et al. [17]. We found there are various reasons for this decrease in TD. One of which is the removal of TD that could be "contagious" – new developers may model their code off existing TD or may directly duplicate poorly written code. Thus, in addition to prioritizing TD that might block key features planned in the upcoming iterations, contagious TD [163] should also be prioritized, especially during times of growth in the development team. If such TD is not removed, it can generate new TD in a vicious spiral, reducing the growth time and compromising the code and culture of the startup in the future.

Encourage autonomy with high-level guidance. We found that in most startups, developers make TD-related decisions with full autonomy. Thus, they could possibly be making poor business decisions without considering the strategic repercussions of their decisions. Providing overall guidance to the developers, so they know what level and types of TD are appropriate can mitigate this risk, while still maintaining developer autonomy.

16.6.2. Strategy to balance TD over time

Startups need to balance several factors affecting the accumulation of TD, to reach a Good Enough Level. However, how do startups do this over time? We report, in Fig. 22, a first interpretation that helps to understand the strategy adopted by the studied cases in different phases.

Fig. 2 shows the accumulation of TD with respect to each startup phase and key events. The black line suggests the accumulation of Technical Debt that has been preferred by the studied startups. We also show GELs ("Good Enough Level"), or else thresholds under which TD needs to be kept via strategic refactorings, otherwise causing possible

المنارة للاستشارات

www.m

disruptive events (red lines and crosses). Finally, in the bottom of the picture, we outline which mechanisms have been reported by the participants to be necessary and effective to keep a GEL of TD in a specific phase.

In the startup phase, startups recklessly accumulate TD. This has been reported to be not only necessary, but very valuable to quickly satisfy the first customers, to reduce risks and costs. However, too much TD can still be disruptive in the first phase, leading to product failure and business disruption, if the acquired TD prevents the successful delivery of the MVP itself. In particular, the cases report that the domain specific technology needs to be well understood and that the usability of the product should not be overlooked (GEL1). In the stabilization phase, a partial refactoring (Stabilization refactoring) is recommended to reach GEL2. In this case, the TD to be prioritized is the one blocking key features planned in the upcoming iterations for the delivery of the product to key customers. In addition, TD that is judged to be especially contagious (likely to spread to the new features and to be picked up by new developers) should be at least considered. The challenges if the startup fails to keep this level of TD is the difficulty (if not the halt) of evolving the system with new features, with the consequent loss of key customers. Additionally, while entering the growth phase, TD that is accessed by new developers can generate new TD in a vicious spiral, reducing the growth time and compromising the code and culture of the startup in the future. Here the high-level guidance and the experience of the developers are key to keep the right level of TD, but a budget needs to be allocated for the refactoring to reach GEL2. During the growth phase, there is a need to remove some more TD (Growth refactoring) to reach a GEL3. If the contagious debt is not removed in the previous phase, it needs to be removed here before hiring new developers. In addition, the code is optimized to be scalable and to be delivered to several customers in the market: the architecture of the system should be refactored to allow the productive management of customer variability, to reduce the cost of maintenance and operations for the developers, to avoid a loss of productivity. In the growth phase, several other mechanisms can be introduced to not only reduce the current TD, but also to prevent the accumulation of future TD (e.g. tools, processes). TD needs to be well communicated in order to make business decisions. In their maturity phase, startups seem to start behaving like mature companies. However, in this study, we do not have enough cases to report common practices related to this phase.

المنارة للاستشارات

www.r

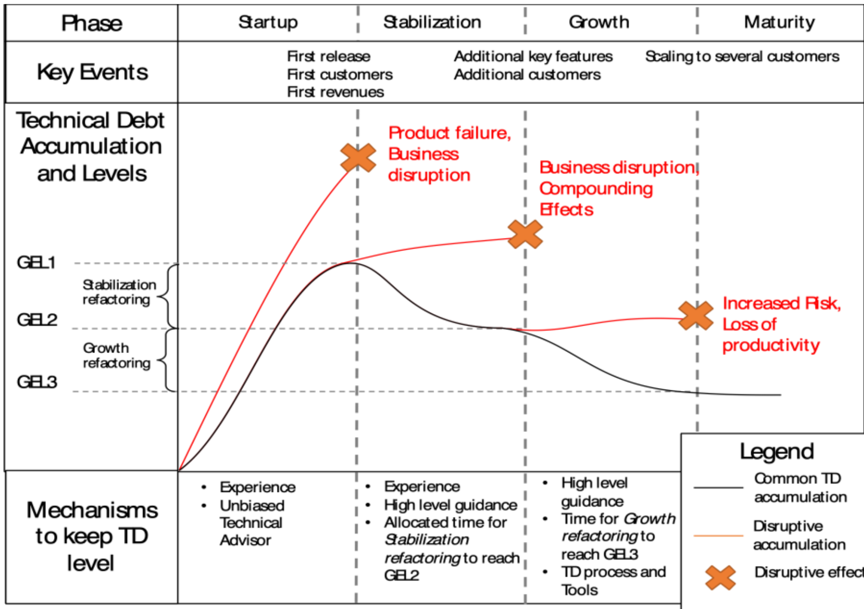


Figure 3. TD balanced differently in different phases

16.6.3. Differences with TD Management in Large organizations

Looking at the current literature, we can see some differences with how startups accumulate and refactor TD, compared to mature organizations.

In both startups and mature organizations, there is often a peak of accumulated TD at the beginning of feature development [110]. However, in mature organizations, there is usually a defined quality threshold, in the form of the desired software architecture or other quality models. In such cases, TD is referred to the divergence from such desired thresholds. Such reference points do not seem to exist in startups. Consequently, they tend to accumulate more TD, which is also considered a benefit. There is, naturally, some level of uncertainty in both startups and mature organizations at the start of a new project. However, the uncertainty in young startup companies is greater than in a mature company [240]. Thus, taking on a right amount of TD seems to be a well-established strategy to deal with the high levels of uncertainty.

Startups rely more on junior developers than in large organizations [241]. Such junior developers are less experienced, and they are less aware of the long-term effects of TD, which consequently leads them to be keener to accumulate it. This choice seems to fit with the importance to accrue TD in the startups. However, as we have seen in all the analyzed cases, an experienced developer (technical lead or CTO) is crucial in the startup team to keep the TD level to desired thresholds. In contrast, in large organizations, most of the team members are expected to have a higher understanding of TD and to make sure that TD is not accumulated [164]. The main constraint is that code developed by large

المنارة للاستشارات

www.r

organizations is continuously integrated with a large codebase and needs to be available and reliable for other teams' work. This is a constraint that does not exist in the startup and stabilization phase of startup companies but comes into play when the startup enters the growing phase.

A similar difference can be seen with respect to processes and tools: a recent survey in the large organization [164] highlights how a third of the participants, answering the survey, use tools to track TD. In startups, we could see the complete lack and conscious avoidance of such processes and tools until the company reaches the growing phase. On the other hand, both in startups and partially (2/3 of the participants) in large organizations [164], we notice the lack of knowledge on how to implement such processes and what tools to use to keep TD at bay. Learning how to manage TD seems to be equally important for large companies and for startups entering the growth phase.

In summary, despite some similarities exist regarding TD management between large organizations and startups, the first three startup phases are fundamentally different from the everyday work in large companies. This is due to the level of uncertainty, the environment, and the business context being very different. In conclusion, the strategic management of TD in startups should not follow best practices related to large organizations. In this study, we are contributing to new knowledge reporting some first recommendations and experiences on how TD is managed in startups.

16.6.3.1. Limitations and Threats to Validity

The main limitations of this study are related to the limited sample of startups investigated and to the qualitative nature of the investigation. However, these are limitations that can be considered acceptable in light of the exploratory purpose of this study. We preferred to gain a deep and rich understanding of the context of a few cases to build a holistic first theory rather than surveying the topic on a high level only. Specific threats to validity include construct validity related to the concept of TD, external validity with respect to the limited contexts analyzed, and reliability of the results affected by the high level of interpretation that both interviewees and researchers might have been injected in the study [73]. To mitigate construct validity, we held a workshop with several of the participants in the startups to clearly define and align on what TD was. We gave concrete examples, we used the up to date definition of TD reported in the Dagstuhl seminar [4], and we asked the participants to share examples in order to test if their understanding matched the community's definition. Additionally, when asking questions, we have always asked and probed the claims by inquiring for additional concrete examples.

To mitigate the external validity threat, we collected information from two different countries in different geographical areas. In addition, the case companies represent different segments, and we interviewed different roles, from developers to CTOs to CEOs, to external advisors. Although we do not claim to provide fully generalizable results in this exploratory study, we have aimed at maximizing the coverage of our cases. Furthermore, we plan to expand our sample in the future, to reach a higher degree of validation of our results.

المنارة للاستشارات

www.r

Reliability threats were mitigated by assuring that two researchers were always present when conducting interviews, that one of the researchers was always attending all workshops and interviews for consistency purposes, and that the analysis was organized in two groups where researchers analyzed the codes separately and then merged the findings. In other words, we made sure that different observers were contributing in different phases of the data collection and analysis, reducing the bias of single researchers.

16.7. Conclusion

This exploratory study set out to provide a first understanding of how software startups reason about TD. Through interviews with 16 software professionals in seven different startup companies, we identified six organizational factors that influence the accumulation of TD in software startups (experience of developers, software knowledge of startup founders, employee growth, uncertainty, lack of development process, and the autonomy of developers regarding TD decisions). We also found that startups must strive towards a Good Enough Level, over time, for their product, while weighing the benefits and challenges associated with taking on TD. This study provides a set of recommendations and a first strategy which can be used by software startups to support their decisions related to the accumulation and refactoring of TD.

المنارة للاستشارات
www.r

www.m

17. Technical Debt in Safety-Critical Software

This goal of this chapter is to explore how the regulation of SCS affects the management and the growth of TD.

In recent years in the software industry, the use of safety-critical software is increasing at a rapid rate. However, little is known about the relationship between safety-critical regulations and the management of technical debt. The research is based on interviews with 19 practitioners working in different safety-critical domains implementing software according to different safety regulation standards. The results are three-fold. First, the result shows that performing technical debt refactoring tasks in safety-critical software requires several additional activities and costs, compared to non-safety-critical software. This study has also identified several negative effects due to the impact of these regulatory requirements. Second, the results show that the safety-critical regulations strengthen the implementation of both source code and architecture and thereby initially limit the introduction of technical debt. However, at the same time, the regulations also force the software companies to perform later suboptimal work-around solutions that are counterproductive in achieving a high-quality software since the regulations constrain the possibility of performing optimal TD refactoring activities. Third, the result shows that technical debt refactoring decisions are heavily weighed on the costs associated with the application's recertification process and that these decisions seldom include the benefits of the refactoring activities in a structured way.

This chapter has been published as:
How Regulations of Safety-Critical Software Affect Technical Debt
T. Besker, A. Martini, and J. Bosch
45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 74-81, 28-30 Aug. 2019.

17.1. Introduction

Safety-critical software (SCS) applications are increasingly affecting our lives and welfare as the amount of software embedded in cars, medical devices, and airplanes are increasing at a rapid rate. The trend points toward using more and more software in safety-critical products. The domain of SCS applications is characterized by regulatory requirements to which companies' software must adhere before they can place their applications on the market.

At the same time as the SCS is being used in critical applications that can endanger people, equipment, and environment [242], these software companies also need to focus on delivering customer value, from a short- and long-term perspective. Similarly to non-safety-critical projects, the safety-critical projects also have to develop software faster and more cost-efficiently and are thereby also incurring Technical Debt (TD).

The TD metaphor was introduced by Ward Cunningham [91] to describe the need to recognize the potential long-term negative effects of immature code that is made during

المنارة للاستشارات

www.m

the software development lifecycle. This debt potentially has to be repaid with interest in the long term. Interest is the negative effect in terms of the extra effort and activities that have to be paid due to the accumulated amount of TD in the system, such as executing manual processes that could potentially be automated or expending excessive effort on modifying unnecessarily complex code, or performance problems due to lower resource usage caused by inefficient code and similar costs [7],[151]. When TD is present in the software, the only significantly effective way of reducing it is to refactor it.

While there are several similarities between non-SCS and SCS, there are also several major differences. One of the most significant differences is that SCS are heavily regulated and require certification against industry standards. These standards have, among other issues, an adverse impact on the process of conducting refactoring tasks. For example, after a refactoring activity in an SCS, the concerned software component needs to be recertified (e.g., retested, revalidated, reverified, etc.) to ensure compliance with the present safety standards. This recertification process requires several extra activities compared to non-SCS, and this process is commonly a very cost- and time-consuming practice. Thus there is a significant potential risk of TD refactoring initiatives being down-prioritized or even proactively avoided with the negative consequence of an increasing and alarming amount of TD in the SCS.

Further, in SCS systems, different components can have different levels of safety regulations, which defines the recertification scope. This means that even if the refactoring changed only a modest part of the software, the recertification cost and effort can, nevertheless, be substantial if the scope of the concerned component is large [243]. A key solution to alleviate this issue is to design a sound architecture and validate it against the criticality of the software [244]. These settings stress the importance of a software architecture that facilitates refactoring with as little effort and cost as possible.

Although there are several research publications addressing TD and some publications related to SCS, to the best of our knowledge, this is the first empirical study focusing on the impact of the SCS standards related to the management of TD in today's software industry.

Our overall research goal is to understand how refactoring initiatives of TD are affected by the SCS regulations. In particular, we are interested in what additional activities are necessary for TD refactorings in response to the requirements of SCS and how these activities influence the decision-making process of TD.

Therefore, we aim to answer the following research questions:

RQ1: What are the consequences and effects of SCS regulations when conducting or planning for TD refactorings?

RQ2: What factors influence the decision-making of TD refactoring in SCS?

RQ3: What software architectural structures contribute to TD refactorings in SCS?

المنارة للاستشارات

www.m

This study makes a novel contribution to research, with respect to the existing body of knowledge on TD, in the following areas:

1. Based on this study's result, we show that there are several consequences of the SCS regulations that need to be addressed when planning for and/or conducting TD refactoring activities in SCS.
2. We present results showing that the effect of these consequences has a significant impact on the SCS in terms of, for example, the amount of TD and the requirement of enhanced implementation guidelines.
3. The results show that when comparing the cost and benefit side of TD refactoring activities in SCS, such analysis has a strong focus on the cost side and pays less attention to the benefit side.
4. This study provides new insights into TD research by revealing that the architectural structure in SCS is of significant importance, where a component- and/or layer-based architecture facilitates and encourages remediation of TD.

The remainder of this paper is structured in seven sections, as follows: In section 2, we discuss related work. Section 3 describes the research methods in detail. Section 4 presents the research results. Sections 5 and 6 discuss the findings and threats to the validity of the study, respectively. Section 7 concludes this study.

17.2. Related work

This section presents related work concerning safety-critical software (SCS) and technical debt (TD).

17.2.1. Definition of SCS

An SCS is a software system the correct operation of which impinges directly on human safety. SCS is defined as: "*systems whose failure could result in loss of life, significant property damage, or damage to the environment*." [245]

17.2.2. Use of Standards and Safety Integrity Levels

SCS is mandated by law or strongly recommended to adhere to region-specific regulatory standards. This means that the software applications are required to pass a regulatory audit process before their placement on the market.

Several international standards guide software companies regarding regional regulations in safety-critical domains and prepare for the regulatory audits. However, these standards are continuously developing [246]. Some standards are general in scope and apply to a broader range of systems, for example, ISO 61508 that covers electric, electronic, and programmable electronic safety-related systems, [247] whereas other standards are

المنارة للاستشارات

www.r

sector- or system-specific, such as IEC 60601 for medical devices, ISO 26262 for functional vehicle safety, and RTCA/DO178B standard for avionics and airborne systems. Within the standards, there are different risk classification schemas. For example, in the ISO 26262 standard, there are four different levels called Automotive Safety Integrity Level (ASIL, level, A to D), which all identify different safety-critical requirements. Likewise, the RTCA/DO178B standard has five different safety levels (DAL, level E to A).

17.2.3. Technical Debt and Safety-critical Software

While delivering on time and within budget are important aspects of every software project, when developing SCS, additional aspects such as reliability and maintainability are also of major importance to avoid high failure costs [248]. Ghanbari [248] states that developers are forced to introduce TD in critical software due to requirement ambiguity, diversity of projects, inadequate knowledge management, and resource constraints. However, his study does not explore the impact the various safety regulatory aspects have on the introduction and growth of TD followed by the possibility of performing TD refactoring tasks.

17.3. Research method

Given the exploratory nature of the study, we performed a multiple case study with four cases. The study includes focus interviews with 20 practitioners specializing in different application domains.

17.3.1. Case selection

Following the classification provided by Yin [66], this study is an embedded case study in which multiple units of analysis are studied in one case [86].

Participating companies were recruited using a non-probability sampling technique [86] of companies developing SCS within our industrial network. Our contact person selected the interviewed professionals from the companies at each company with the goal of inviting several different roles having different levels of responsibilities and experiences with SCS.

A brief description of the participating companies and respondents is presented below and summarized in Table I. On average, the interviewees had 17 years of experience in software engineering in general and 12 years working specifically with SCS. For confidentiality reasons, the companies have been anonymized in this study.

Company A is a large company within the automotive industry located in Sweden with many sites or daughter companies worldwide. In their product domain, SCS plays a key role and will become even more important and critical in their future product release plan. The company is subject to the ISO 26262 standard. All SCS are internally assessed for

المنارة للاستشارات

www.r

consistency with their current standards before going into production. This company has worked actively with remediation initiatives of TD for several years and has, in general, a high awareness of the negative consequences of TD.

Company B operates within the aeronautics segment where a major share of the developed software is characterized as safety-critical and applies commonly to the highest safety-critical level within this domain. The company bases its development process around the RTCA/DO-178B standard. All SCS are internally assessed and certified before going into production. This company has, in general, a high awareness of the negative consequences of TD although their TD management process and goals are not explicitly stated within their organization, and they have not dedicated a certain amount of budget to remediation of TD.

Company C is a leading consultant company within the SCS domain in Scandinavia today. The company generally operates within several different business areas, but the interviewees in this study mainly work with safety-critical applications within the defense and aeronautics segment, developing software under the RTCA/DO-178B standard.

TABLE I - CHARACTERIZATION OF RESPONDENTS

CompId / RespId	**Selection of Respondents**		
	Role	***Experience with SCS***	***Software Experience (incl. SCS)***
CA / R1	Solution Architect	18	25
CA / R2	Software Architect	2	17
CA / R3	Product Manager	25	25
CA / R4	Solution Architect	20	21
CA / R5	Software Architect	20	33
CA / R6	Software Integration Architect	2	13
CA / R7	System Safety Leader	10	24
CA / R8	System Safety Engineer	3	4
CA / R9	Solution Architect	5	10
CB / R10	Line Manager	7	20
CB / R11	Material Group Manager	19	19
CB / R12	Software Architect	21	30

المنارة للاستشارات

www.r

CompId / RespId	Selection of Respondents		
	Role	***Experience with SCS***	***Software Experience (incl. SCS)***
CB / R13	Project Leader	6	12
CB / R14	Software Architect	10	12
CB / R15	Product owner	12	14
CB / R16	Funcional Safety Technical Leader	3	5
CB / R17	Tester	12	20
CB / R18	Tester and Developer	3	3
CC / R19	Responsible for the technology of safety-critical system engineering	21	6

17.3.2. Research design

As illustrated in Fig. 1, this study's research design was divided into five phases. The following sections describe these phases and the related research methods used in each stage.

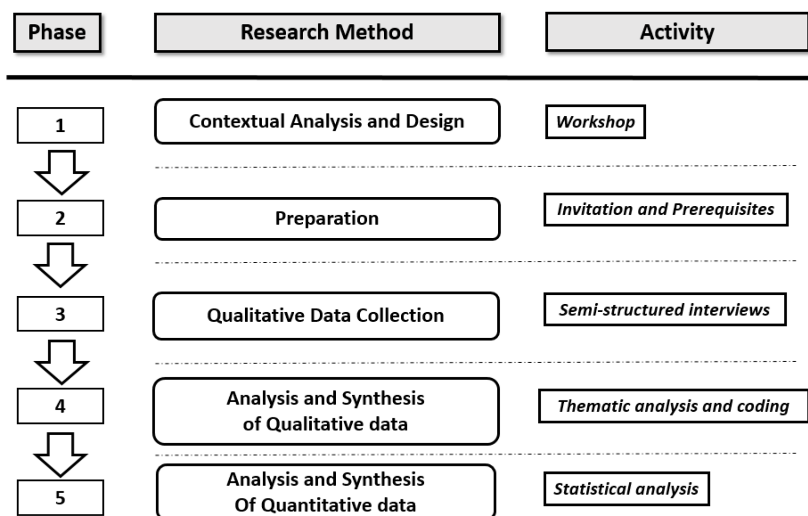


Fig.1 Visualization of the research model and method and activity used in each phase.

المنارة للاستشارات

www.r

17.3.2.1. Contextual Analysis and Design

First, the study was presented in a cross-company workshop with managers from the participating companies. The goal of this workshop was to explore the area of TD in SCS applications briefly and to define the overall aim and scope of this study.

17.3.2.2. Preparation

Following the guidelines of [67], we emailed an invitation letter describing the study to those 19 interviewees who had agreed to participate in the study. The letter included educational material intended to minimize inter-observer and inter-instrument variability. All distributed educational material has been made available at https://doi.org/10.6084/m9.figshare.7725857.v1 .

17.3.2.3. Qualitative Data Collection

In the third phase, we employed semi-structured interviews in which the interview guide was defined in advance, but the questions not necessarily asked in the same order, allowing flexibility to explore interesting insights as they emerged. Examples of interview questions are presented in Table II and grouped by the corresponding research question.

The interviews were conducted between January and February 2019. Each interview lasted between 60 and 90 minutes. To obtain a more accurate rendition of the interviews, all interviews were digitally recorded and transcribed verbatim. All interviewees were asked for recording permission before starting, and all interviewees agreed to be recorded and to be anonymously quoted for this paper. The interviewer also informed the interviewees that participation was voluntary and that the interviewee could cancel the interview at any time.

TABLE II - EXAMPLES OF INTERVIEW QUESTIONS

RQ	Examples of Interview Questions
1	• What extra activities need to be done after a TD refactoring (compared to non-SCS)? • Do you ever consider avoiding or postponing a TD refactoring activity and instead pick another solution, like a work-around? • What is the main reason for doing work-arounds?
2	• Do you do a cost-benefit or a change impact analysis (or something else) before making TD refactoring decisions? • Do you have any kind of report template describing the possible change before making the decision of refactoring TD or not? • Do you think that you would refactor more TD if it were for the SCS regulations?

المنارة للاستشارات

www.r

RQ	Examples of Interview Questions
3	• Please describe the structure of the different safety levels in your software architecture briefly? • Do you think that the architectural design is different in your software due to being an SCS (and in what way)? • From a TD refactoring perspective, do you think your software could be designed in another way to make the TD refactoring more easily and cost-efficient?

17.3.2.4. Analysis and Synthesis of Qualitative data

The acquired qualitative information collected in phase 3 was analyzed using thematic analysis [84] to identify, analyze, and report patterns and themes within the data. The thematic analysis was conducted using the following five steps.

1) The audio-recorded qualitative data collected from interviews were transcribed, and we familiarized ourselves with the data through careful reading of the transcripts to acquire a holistic overview of the content.

2) The initial codes were constructed from the data, and the data were organized into distinct groups using a thematic map [249]. To assist this phase of the analysis, a qualitative data analysis (QDA) software tool called Atlas.ti was used.

3) The themes were identified by organizing the codes into potential themes and collecting the relevant coded data extracts within each theme. Each extract of data was assigned to at least one theme and, in many cases, to multiple themes. For example, the citation "we track every cost of requirements, verification tests, so everything has to be tracked to the top level design" was coded as "Cost of Refactoring" in the theme "Decision-making of TD refactoring."

4) This phase focused on identifying the essence of each theme and determining what aspect of the data is captured by each theme. This phase also stressed the importance of not just paraphrasing the content of the data extracts but also identifying what is interesting about them and why.

5) The final phase of the thematic analysis took place when we had a set of fully developed themes; it involved the final analysis and write-up of the publication.

Fig. 2 shows a subset of the result of the analysis process where the mapping between different hierarchical code levels and the taxonomy are illustrated. Due to space limitation, this figure represented a curtailed and reduced part of the complete data collection map.

المنارة للاستشارات

www.r

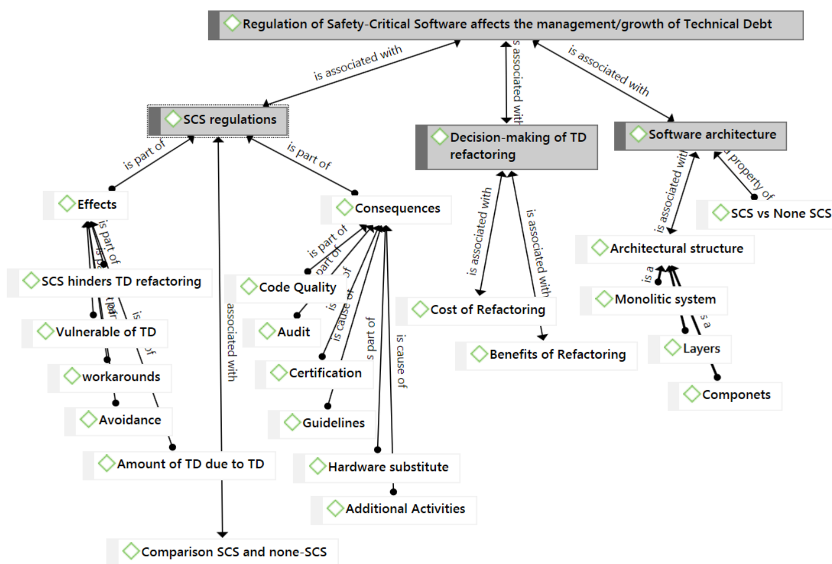


Fig. 2 Subset of Coding Scheme

17.4. Results and findings

The following subsections present results for the research questions presented in Section I and are grouped according to each research question.

17.4.1. Consequences and Effects of SCS Regulations when Conducting or Planning for TD Refactoring Activities (RQ1)

Our analysis identified several factors that, due to SCS regulations, are affected when planning and/or conducting TD refactoring activities. Our result indicates that TD refactoring of SCS requires several additional activities and costs, compared to non-SCS. To fully understand the consequences of the SCS regulation on TD refactoring activities, we first describe the background of the consequences and thereafter we present their effects.

The consequences of the SCS regulations on TD refactoring activities are grouped and summarized into the following themes:

17.4.1.1. Additional documentation

Before performing a refactoring activity, there is often an additional need to create new safety cases with new safety analysis to show that the SCS regulations properly adhere. There is also an extensive amount of documentation describing the source code and its dependencies that need to be updated after a refactoring activity. This situation can

المنارة للاستشارات
www.r

introduce additional documentation debt, and one interviewee describes this, “We can change the code if it's really important, but you don't change it in the complete documentation, because the documentation has a hundred references down to other documents.”

17.4.1.2. Additional simulations/emulation

After a TD refactoring activity, the altered application commonly needs to be tested and run in simulation and/or an emulation environment. This activity is both time-consuming and potentially dependent on resources from other departments. This activity also commonly comes with additional license costs for using this equipment and tools. This issue was mentioned by one of the interviewees, “One of the biggest challenges performing refactorings in safety-critical software refers to the testing since sometimes you have limited access to the equipment that you need, and that can slow you down for weeks if a lot of people need the same shared equipment…and these equipments can cost billions of Swedish crownes”.

17.4.1.3. Additional risk assessment

Refactoring of SCS activities requires additional risk assessment due to potential time delays, since the complete refactoring process includes and is dependent on staff and resources from several other departments, and not only the staff who performed the initial refactoring task. This dependency on external resources entails a reduced control over the strategic planning and management of time.

17.4.1.4. An extended test scope

Testing refactored SCS often requires not only testing the specific isolated refactored component; the test scope is also commonly extended to related components/subsystems This is commonly referred to as “freedom from interference” and refers to that an alteration in one component will not lead to a fault in another more safety-critical component. One interviewee from company A it this way: “In safety-critical software, the freedom of interference is very important. Meaning that you say to someone, ‘If I do this, does it affect anything else?’”

There are also refactoring tasks that potentially can lead to the need for retesting of the complete application. One interviewee from company A describes the test process of a complete application: “At a minimum, you have to drive some laps of 100,000 km to make sure that the system [a vehicle] is working as it was before [the refactoring].”

17.4.1.5. All TD refactoring activities are concerned

The interviewees described that the additional activities and costs apply not only to major refactoring activities; they commonly also apply to minor refactoring tasks. This implies that even minor refactoring activities could cause significant costs and activities, and

المنارة للاستشارات

www.r

often the practitioners could not perform refactoring activities without assistance or authorization from colleagues or managers.

17.4.1.6. Requires recertification or requalification

Another consequence with the significant impact of a TD refactoring is the requirement of recertification or a re-qualification of the concerned part of the SCS. This process commonly involves an internal audit of the concerned components and its related components due to dependencies among them. This process was described as very expensive, laborious, and time-consuming for the projects but also dependent on the involvement of staff and resources outside the direct software development projects. One manager at company B addressed this issue: "Usually, when we have the certified software, it is rare that we go back and do refactoring of technical debt. We try to have all those discussions before the certification, and usually, the reason is the cost of the recertification process and the effects of other related software that also would require recertification".

The effects of the consequences are grouped and summarized into the following themes:

17.4.1.7. Proactive guidelines

Since the refactoring of TD in SCS requires both extra activities and costs, it is of vital importance to implement SCS solutions as optimally as possible from the very beginning. This was addressed by a proactive mindset by all the companies in this study, and this strategy influenced all the different parts of the development process. For instance, an interviewee from company A described that, due to the SCS regulations, their development process explicitly uses reinforced development guidelines designed specifically to reduce the introduction of TD and thereby proactively reducing a future need of performing TD refactoring tasks: "We wanted to build a high-quality system in general, but we put extra focus on the guidelines based on the safety-criticality of the software." A similar strategy was also used at company B where their implementation guidelines included customized advice and rules on how to implement the software according to each of the concerned safety-critical levels, and they also have a strong focus on the review process of the implementation: "So, as a developer, you're not allowed to make your own decisions on how to do software. It's reviewed, and the source code is reviewed by many people many times." This view was echoed by another respondent at the same company, "Our code guideline is very strict, and compared to non safety software, our guideline is much stricter….A lot of the rules are there in the guideline to enforce, for example, compiler behavior and stuff like that."

17.4.1.8. Avoidance of refactoring activities

As a result of the above-mentioned consequences of performing TD refactoring, several interviewees describe that, due to extra activities, TD refactoring activities are commonly

المنارة للاستشارات
www.r

deliberately avoided with the result of an increasing amount of TD in the software. As one interviewee at company A expressed, “We have one case right now where, if we refactor a lot in our present functionality [as we would like], we need to do a new field test in order to prove that we have the right ASIL level and assessments and so on. So, we avoid doing it right now because we neither have the time nor resources to re-evaluate the new solution.”

17.4.1.9. Work-around solutions

Our result also shows that practitioners often develop work-a-round solutions to avoid TD refactoring activities and thereby to avoid the associated consequences and the need for the additional activities and costs presented in section IV.B.1. An interviewee at company A said, “Work-around [to avoid performing TD refactoring] is very, very common, to be honest.” Commonly, these work-around solutions were done directly in the software codebase (in a way that required fewer additional activities or recertification).

There were also situations where the solution was strategically and deliberately done in a suboptimal way by implementing a work-around solution in a component with a lower safety-critical level compared to the original component. The main reason for this was the high difference in development costs and efforts due to the different SCS levels. One interviewee from company A explained the difference in required working time of development due to different ASIL levels: “The different [required working time] between ASIL A, and ASIL B is a factor of 10, between ASIL B and C is also a factor of 10, and between ASIL C and D is a factor of 10. This means that it takes 1000 more time to go from ASIL A to ASIL D.”

Performing work-around solutions totally outside of the certified SCS scope of the application was sometimes a common practice to avoid additional refactoring tasks, where these work-around solutions were characterized as poor architectural design decisions by adding a significant amount of architectural TD to the software. One interviewee also described a work-around situation in which the end user of the software had to perform extra tasks manually when using the application due to the company’s preference not to refactor TD in the code: “They have instructions for the end user to handle certain technical debt in the code because it is too expensive to fix it.”

By doing these suboptimal work-around solutions, the architecture was described as inheriting additional TD continuously. The work-around solutions were commonly documented less and created difficulties for future architecture since these work-around solutions commonly generated additional future work-arounds as the work-arounds were more difficult to refactor. One interviewee described this way: “In some areas, there is software that nobody understands anymore, so you don’t want to alter things there and make even more [a] mess of it.”

المنارة للاستشارات

www.r

17.4.2. Factors Influencing the Decision-Making of TD Refactoring in SCS (RQ2)

As mentioned in section IV.A, TD refactoring in SCS is associated with several extra costs and activities and requires, thereby, for example, additional budget, resources, time, and a need for recertification or a requalification process (of the software and/or the hardware). This infers that the overall cost equation formula for TD refactoring activities for SCS becomes more complicated and extended compared to non-SCS.

None of the investigated companies explicitly performed a cost-benefit (or equivalent) analysis in a structured way, comparing the benefits of performing the TD refactoring with the cost of it. The main reason for not conducting such analysis was described in terms of a lack of data/estimates on the benefit side of the equations (and not in terms of lacking information on the cost side) both from a direct (from the current situation) or an indirect (future-oriented) perspective.

As one interviewee said, "We know the costs, but we don't know the benefits because the benefit is usually effected by the kind of feature updates that will come in the future, and we don't know this and how much time and effort we would save in that case." The lack of knowledge about how to calculate the interest cost for the TD items was expressed as being one major obstacle to performing such evaluation. On the other hand, calculating the overall costs of refactoring TD were not generally considered as difficult or cumbersome. By having only data/estimates on the cost side and lacking the equivalent information on the benefit side was described as heavily influencing the decision-making of whether to invest in TD refactoring activities or not.

Taken together, our result shows that the TD refactoring decisions are made with cost-based optimality conditions rather than comparing both the cost and benefit side of the equation. Several of the interviewees stated that it would be helpful to motivate the need for investing in TD refactoring activities if they had access to data/estimates on the benefit side. The interviewees also believed such information would lead to more TD refactoring initiatives being taken, and thereby the amount of TD in the SCS could be reduced.

17.4.3. Software Architectural Structures Contributing to TD Refactorings (RQ3)

There are different architectural structures such as component-based, pipes and filters, monolithic, and layered structures [250]. Our analysis shows that specifically the architectural structure of SCS has a huge impact on the possibility and probability of performing TD refactoring activities. SCS based on monolithic architecture components is characterized by interconnected and interdependent components and were described as a major hindrance for TD refactoring tasks in SCS due to the interwoven nature of different components or resources. Several interviewees stated that it is specifically important in SCS to avoid the monolithic structure since performing refactoring activities in such architectural settings was described as requiring a huge scope of the software to

المنارة للاستشارات

www.m

be recertified or requalified and thereby the remediation of TD were often postponed, avoided, or managed by doing work-arounds instead.

On the other hand, a modular SCS architecture that structures the application as components using separations and encapsulation strategies of component-based structures of loosely coupled units or layer-based structures (or both), were described as contributing to increased likelihood of TD refactoring tasks in the SCS domain to be performed. One interviewee describes their architectural strategy this way: "We try not mixing different concerns and keeping the complexity low by identifying and isolating the criticality of different domains, using small modularized pieces and keeping the monolithic approach out."

One of the most important goals of the architectural design was described in terms of keeping the overall complexity low by separating the software component while considering the hardware implementation at the same time. One interviewee expressed this goal this way: "Keep your life-saving safety-critical parts in a separate piece of hardware, with a separate process that monitors function etcetera, and to keep the big box for what it is good at like good for multi-purposing or multi-simulation or concurrent execution of things, like cloud or machine learning, etcetera." Such structures were described as enabling the TD refactoring to be performed affecting a less extensive and a limited area of the software, thereby making the recertification process more manageable.

The majority of the interviewees described their software architecture as heavily influenced and tailored by their SCS regulations, and, therefore, the architecture would have been designed differently if the software did not have to adapt to the safety standards.

One interviewee said of this difference, "Our middle layer in the architecture would have looked different [if it was not SCS] since the intention of the decision level is actually to abstract and isolate different ASIL levels because it would be quite hard and expensive to maintain these dependencies otherwise."

17.5. Discussion

The result reveals that the SCS regulations have a significant impact on the overall development process and specifically when planning and/or conducting refactoring of TD. In this section, we discuss the result for each of the stated research questions.

17.5.1. Consequences and Effects of SCS Regulations when Conducting or Planning for TD Refactorings

The first research question (RQ1) addressed the consequences and effects of SCS regulations on TD remediation tasks. The results of this study show that SCS regulations have an immediate and adverse impact on the refactoring activities of TD. Due to additional costs and the need for additional activities due to the regulations, practitioners

المنارة للاستشارات

www.r

frequently avoid performing refactoring activities, with the consequence of introducing additional TD in the form of, for example, suboptimal work-around solutions.

One interesting finding is that this negative effect was quite known to the companies and they had, therefore, created proactive strategies for implementing the software more carefully with higher quality from the very beginning.

17.5.2. Factors Influencing the Decision-Making of TD Refactoring in SCS

The second research question (RQ2) in this study sought to explore the influencing factors which play a role when deciding TD refactoring initiatives. The associated negative consequences of refactoring activities need to be addressed when making TD refactoring decisions due to the need for additional activities, additional resources, and also the need for additional working time. However, because only the information on the cost side of a cost-benefit analysis was known to the professionals making the TD refactoring decisions, there is an imminent risk of overestimating the costs of a TD refactoring activities and underestimating the costs and the negative effects of postponing or ignoring the presence of TD. The investigated companies all expressed a lack of a quantitative and/or a qualitative guideline to assist them when making this kind of decision. This unbalanced cost-benefit equation could potentially cause suboptimal refactoring decision is made.

17.5.3. Software Architectural Structures Contributing to TD Refactorings

The third question (RQ3) in this research focused on different architectural structures and their corresponding impact on refactoring tasks from an SCS certification perspective.

The structure of the SCS software application consists commonly of several different subsystems, which in turn consist of several different qualified or certified components. Both the subsystems and the associated components can potentially be of different safety-critical levels, causing different types of recertification activities after refactoring of the software.

An obvious finding to emerge from the analysis in this study is that the architectural structure of the software has a significant impact on the remediation of TD task in SCS, where a monolithic architecture complicates the TD refactoring activities.

However, even if these findings suggest that a component and/or layer-based type of architectural structure is, in general, beneficial when reducing the scope of the refactoring actions, the findings also point to this structure as enabling work-around solutions by instead altering components that require less or no recertification activities.

المنارة للاستشارات

www.r

It is evident from the findings that a component- and/or layer-based architecture facilitates and encourages remediation of TD, but on the other hand, such structures also contribute to making suboptimal work-around solutions to avoid optimal TD refactorings possible.

17.5.4. The Counterproductiveness of the SCS Regulations

Independently of domains or regulations, all SCS is heavily regulated and requires certification against industry standards. One of the main goals of SCS regulation refers to the accuracy and consistency of the software where the overarching goal is to deliver high-quality software. Gauging these regulations incorrectly could cause compliance failure and the creation of a flawed software application consequently escalating costs for the companies and the potential for loss of life for the user.

However, even if these regulations have the best intention to produce a high-quality software product, the findings in this study demonstrate that these heavy regulations are conceivably counterproductive for the achievement of this goal since they potentially can constrain the possibility of performing optimal TD refactoring activities efficiently.

Due to the lack of information about the interest cost of the TD (as described in section V.B), there is a potential risk of implementing a work-around solution or postponing the refactoring task instead of an optimal refactoring activity.

This study's findings suggest that the SCS regulations have, in several cases, the opposite effect of what was intended and desired; instead of promoting refactoring activities to raise the software quality, the regulations contribute to the further introduction of TD and thereby potentially decrease both the maintainability and evolvability of the software.

Fig.3 exemplifies this potential counterproductiveness of SCS regulation and how implemented suboptimal work-around solutions contribute to a recursive loop (dashed line in red) of additional TD followed by a decrease of both the maintainability and the evolvability of the software.

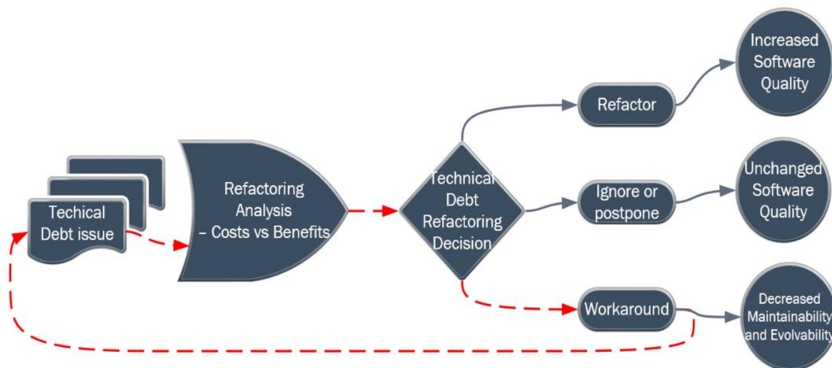


Fig.3 Visualization of counterproductiveness of the SCS regulation

المنارة للاستشارات
www.r

17.6. Limitations and threats to validity

The limited sample of companies investigated (three) and the qualitative nature of the investigation are the main limitations of this study. However, these are limitations that can be considered acceptable in light of the exploratory purpose of this study. Our goal was to gain a deep and rich understanding of the context of a few cases to build a holistic theory rather than surveying the topic on a high level only. The validity of the study includes construct validity related to the concept of TD, external validity with respect to the limited contexts analyzed, and reliability of the results affected by the high level of interpretation that both interviewees and researchers might have injected into the study [73]. To mitigate construct validity, we held a workshop with the participating companies to clearly define the scope of the study, and before conducting the interviews, education material was provided in which we presented definitions of TD from, for example, the Dagstuhl seminar [4]. To mitigate the external validity threat, we collected information mainly from two different safety-critical domains. Also, the case companies used different safety regulation standards, and we interviewed several different roles, from developers to testers, to architects and safety technical leaders. Although we do not claim to provide fully generalizable results in this exploratory study, we have aimed at maximizing the coverage of our cases. Furthermore, we plan to expand our sample in the future, to reach a higher degree of validation of our results.

Reliability threats concern whether the analysis and the results depend on the involved researchers [73]. In this study, we have followed strict protocols and guidelines for analysis which also is reported in the study. Additionally, all findings were derived by the first author of the study and after that reviewed by the two other researchers, which we argue limits the reliability threat.

17.7. Conclusion

This study set out to explore how the regulation of SCS affects the management/growth of TD. SCS must adhere to domain-specific regulatory frameworks. This means that these software applications are required to pass a regulatory certification process prior to being placed on the market.

This study has identified that these SCS requirements pose numerous challenges when conducting or planning for TD refactoring activities since there are several negative consequences and effects associated with altering the software. The main consequence of a TD refactoring activity is related to the costs and efforts of the recertification process, with the potential risk of either ignoring the refactoring task or performing a suboptimal work-around solution instead. The results of this study, therefore, indicate that the heavy SCS regulations potentially can force the companies to perform work-around solutions that are counterproductive in achieving a maintainable and evolvable software since the regulations constrain the possibility of performing optimal TD refactoring activities. The process for making TD refactoring decisions was strongly influenced by and related to the cost of performing the refactoring and seldom based on the benefits of it. Our result

المنارة للاستشارات

www.r

also demonstrates that the architecture of the SCS is important in order to facilitate and encourage the remediation of TD, where a component and/or layered architecture were preferred over a monolithic architecture.

المنارة للاستشارات

www.r

18. Prioritization of Technical Debt in Backlogs

The goal of this chapter is to assess how the prioritization of TD is carried out in practice by practitioners in today's software industry.

Remediation of technical debt through regular refactoring initiatives is considered vital for the software system's long and healthy life. However, since today's software companies face increasing pressure to deliver customer value continuously, the balance between spending developer time, effort, and resources on implementing new features or spending it on refactoring of technical debt becomes vital. The goal of this study is to explore how the prioritization of technical debt is carried out by practitioners within today's software industry. This study also investigates what factors influence the prioritization process and its related challenges. This paper reports the results of surveying 17 software practitioners, together with follow-up interviews with them. Our results show that there is no uniform way of prioritizing technical debt and that it is commonly done reactively without applying any explicit strategies. Often, technical debt issues are managed and prioritized in a shadow backlog, separate from the official sprint backlog. This study was also able to identify several different challenges related to prioritizing technical debt, such as the lack of quantitative information about the technical debt items and that the refactoring of technical debt issues competes with the implementation of customer requirements.

This chapter has been published as:
Technical debt triage in backlog management
T. Besker, A. Martini, and J. Bosch
International Conference on Technical Debt, Montreal, Quebec, Canada, 2019, pp. 13-22, 2019

18.1. Introduction

Technical debt (TD) is a metaphor introduced by Ward Cunningham [91], where TD is described as a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or even impossible [4].

Commonly, software companies need to consider the tradeoffs between the overall quality of the software and the costs of the software development process in terms of required time and resources.

Examples of this tradeoff can be seen in scenarios in which companies deliberately or undeliberately implement sub-optimal solutions to shorten the time-to-market or when resources are limited in practice. However, if TD is left unchecked in the software, TD can lead to large cost overruns, causing high maintenance costs due to internal software quality issues [95] and the inability to add new features [50]; it may even lead to a crisis point where a huge, costly refactoring or a replacement of the whole software needs to be undertaken [121]. This stresses the importance of regularly refactoring initiatives in the software. However, even if the best intention is to refactor TD as soon as possible after it

المنارة للاستشارات
www.r

has been identified or implemented, there is a tendency toward postponement of these refactoring tasks since, commonly, there are other important deadlines in the near future, where these refactoring tasks are often down-prioritized in favor of implementing new features [121].

The decision-making about if and when a TD item should be refactored is commonly performed as a part of the backlog management (BM) process, where each TD item is assessed and prioritized based on several different criteria. Thus, the BM process is important to facilitate the decision-making process to keep the level of TD at bay; it is also important to understand which factors influence these decisions and what challenges are faced during the prioritization of which TD items to refactor [164].

In general, TD has been extensively studied, and there are studies addressing and giving a theoretical recommendation about the prioritization process of TD. However, less attention has been paid to how the prioritization process of TD is carried out in practice in today's software industry. So far, only very few studies [149], [164] have investigated how the prioritization of TD is practically carried out in today's software industry.

To the best of our knowledge, this is the first empirical paper focusing on how the prioritization of TD is practically carried out in today's software industry together with an assessment of which issues are affecting the process in today's software industry.

This study aims to understand how software companies prioritize TD within their product BM process. In particular, we are interested in how the prioritization process of TD is carried out in practice and understanding which related factors influence the process together with potential challenges.

We, therefore, aim at answering the following research questions:

RQ1: How is the prioritization of technical debt carried out, and what factors influence the process?

This research question will explore how the prioritization process is carried out in practice and to what extent gut feelings influenced the decisions and whether they are managed proactively or reactively.

RQ2: What are the challenges of prioritization of technical debt?

This research question aims at identifying challenges related to the prioritization process of TD.

The remainder of this paper is structured in seven sections, as follows. Section 2 introduces the background related work. Section 3 describes the research methodology. Section 4 presents the research results. Sections 5 and 6 discuss the findings and threats to the validity of the study, respectively. Finally, Section 7 concludes this study.

18.2. Research model and background

Initially, we performed a literature review by searching for research publications addressing the prioritization process of TD. Based on the outcome of this review, we have a formed a conceptual model as presented in Fig.1. In total, five different dimensions

المنارة للاستشارات

www.r

were identified which are particularly interesting when portraying how the TD prioritization process is carried by practitioners and what related factors influence the process.

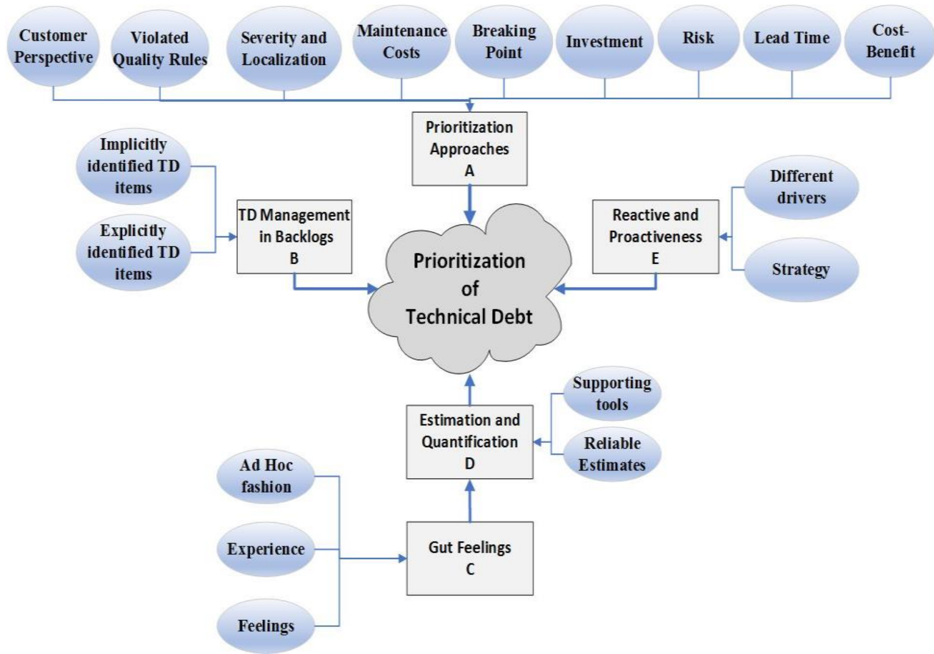


Fig 1. Conceptual Model — Prioritization of Technical Debt

18.2.1. Different approaches to prioritizing TD

From a theoretical perspective, the prioritization process of TD is relatively well represented by several academic studies. However, there is a current paucity of empirical research focusing on how the prioritization of TD is carried out in practice.

Numerous papers propose different prioritization approaches to assist the decision-making process when prioritizing TD [48], [251], [49], [50], [51], [42], [52].

Several of these publications investigate different aspects of the TD that need to be considered during the process. Frequently, cost-benefit analysis is described related to the prioritization process [50], but also other perspectives are mentioned. For instance, Falessi and Voegele [252] focus on the prioritization of violated quality rules. Other criteria that are proposed when prioritizing TD are, for example, the payment of debt items based on the impact the TD has on the software from a customer perspective and the nature of the debt in terms of its severity and localization [49].

In another study, Chatzigeorgiou et al. [253] adopt a prioritization strategy focusing on the timing of repaying the debt in terms of identifying a breaking point where the interest of the TD items reaches the level of the corresponding principal. Further on, Snipes et al. [254] highlight the importance of developing a cost-benefit analysis that also incorporates the financial aspects of the decision aspects during the prioritization of TD.

المنارة للاستشارات

www.r

Guo and Seaman [30] adopt a different decision-making strategy determining the optimal collection of TD items that should be incurred or held, from the perspective of incurring TD as an investment. Also, the need for taking the business perspective into account when prioritizing TD is discussed in a study conducted by Reboucas et al. [255]. According to another study carried out by Martini and Bosch [121] addressing different types of information needed by project owners and architects during the prioritization activity of architectural TD (ATD) shows that vital aspects, such as lead time, maintenance costs, and risk are important factors to consider during the prioritization process.

The major difference between the above-listed studies and ours is that we focus our study on how the prioritization process of TD is carried out in software companies in practice.

18.2.2. The presence of TD items in backlogs

Schmid [119] recommends that TD items should be identified in a TD backlog, and a study by Ernst al. [16] surveying 1831 software engineers and architects found that 31% of the respondents in the survey stated that TD was an implicit part of their backlog and 25% stated that TD was an explicit part of their backlog. Unfortunately, it is not clear in that study how the explicitly and the implicitly identified TD items differ. In comparison to that study, our study aims to understand what different types of backlogs the TD items are identified in and how they differ in management and during the prioritizing process. In our recent study [164], we concluded that TD is either documented in a dedicated backlog for TD issues or in a feature backlog where TD items are mixed with features.

18.2.3. The influence of gut feeling

Both [56] and [52] state that decisions related to TD are largely based on a manager's gut feeling, rather than hard data gathered through appropriate measurement. This notion is echoed by [53] who state that decisions related to cost and scheduling of architectural TD are often done in an ad hoc fashion, based largely on the experience and gut feelings of the architects.

However, to the best of our knowledge, no study addresses the influence of gut feeling on the prioritization process of TD for refactoring.

18.2.4. Estimation of the value of refactoring TD

Cost and value estimations of refactoring initiatives are essential for managing TD since these estimations assist in planning the work processes and also prevents potential cost and schedule overruns. However, making cost and benefit decisions regarding TD is challenging [53], and several authors highlight the difficulties of obtaining such reliable estimates [53], [54], [55], and there are only very few supporting software tools available to assist such activities.

المنارة للاستشارات

www.r

18.2.5. Reactive- and proactiveness of TD management

A reactive approach refers to what currently is happening to the present project, whereas a proactive approach focuses mainly on how to improve future projects [56]. When making decisions related to prioritizations of TD refactoring initiatives, these decisions can take a reactive and/or proactive approach.

In general, TD management is considered to be largely reactive since it often must cause significant pain on multiple fronts before it is addressed [16]. [110] state that refactorings are often overlooked in prioritization and they are often triggered by development crises in a reactive fashion.

However, none of these studies examines whether the prioritization of TD is driven by a strategy that is reactive or proactive.

18.3. Research Methodology

The goal of this study is to understand how software companies prioritize TD within their product BM process. To achieve that, we chose a mixed research methodology by acquiring both quantitative and qualitative data by performing a multiple case-study on four software-developing companies. The study includes focus interviews with 17 practitioners specializing in different application domains. Additionally, we collected data using a survey (n=17).

18.3.1. Case Selection

Following the classification provided by Yin [66], this study is an embedded case study in which multiple units of analysis are studied in one case [86]. The sample population was selected using a non-probability sampling technique [86], where the selection of participant companies was obtained using convenience sampling. The selection of participants was conducted by first reaching out to contact persons to whom we had direct access within our network. After describing the study, this person then suggested colleagues with knowledge about the companies' prioritization process who were invited to participate in the study. A brief description of the participating companies and respondents is presented below and summarized in Table I. For confidentiality reasons, the companies have been anonymized in this study.

TABLE I - COMPANIES AND RESPONDENTS

Company Description			
ID	***Application Domain***	***Company Size***	***Roles***
A	Telecommunications	Large	• Manager Architecture Unit • Program Manager • Product Guardian • Software Architect

المنارة للاستشارات
www.r

Company Description			
ID	***Application Domain***	***Company Size***	***Roles***
B	Automotive	Large	• Solution Lead • Delivery Leader • Solution Architect • IT Architect • Business Analysis
C	Mobility Solutions	Large	• Software Project Manager • Technical Team Leader • Software Architect
D	Packaging solutions and food processing	Large	• Developer • Line Manager • Software Engineer • Software Engineer • Developer

Company A is located in Sweden, and they are developing software using the Scaled Agile Framework (SAFe). Using this approach, their BM approach includes the use of a hierarchical backlog structure in terms of Epics.

The software architects have dedicated board meetings on a weekly basis where (among other things) they prioritize technical debt. This company has worked actively with remediation initiatives of technical debt for several years and has, in general, a high awareness of the negative consequences of technical debt.

The interviewees work both with legacy software with a maintenance focus and with newer software having a strong customer-requirement focus.

Company B is located in Sweden. Their BM process is synced among different development teams within the same department in the same country.

The company adopted an agile software development approach where a BM process is used for planning and prioritizing the development activities within each iteration. This company has an overall backlog which is broken down into several minor team backlogs, which are managed by each team.

The interviewees from this company represent both managers who work with the overall backlog (team A) and developers working with the team backlogs (team B). The items in the backlogs have a direct connection to external customer requirements.

Company C is located in Sweden, but their head office is located in another European country. The overall BM process is synchronized between the different countries. The company uses an agile software development approach where a BM process is used when prioritizing the development tasks. The interviewees from this company represent two different development teams from two different departments. The first team A develops software which is mainly a part of an internally used platform, and thought does not have an external (outside the company) customer. This teamwork mainly on legacy software, where maintenance tasks are most common. The second team B works directly towards

المنارة للاستشارات

www.r

an external customer, where the user stories are based on the customer requirements. This team implements continuously new features.

Company D is located in Sweden, but the company has sites in several parts of Europe. The company supplies complete systems for automation services with extensive in-house software development of embedded, real-time software systems. The overall BM process is synchronized between different development teams, located in different countries, and sharing the same sprint backlog. The software development is today strongly driven by requirements from external customers.

18.3.2. Research Design

As illustrated in Fig.2, this study's research design was divided into five phases. The following sections describe these phases and the related research methods used in each stage.

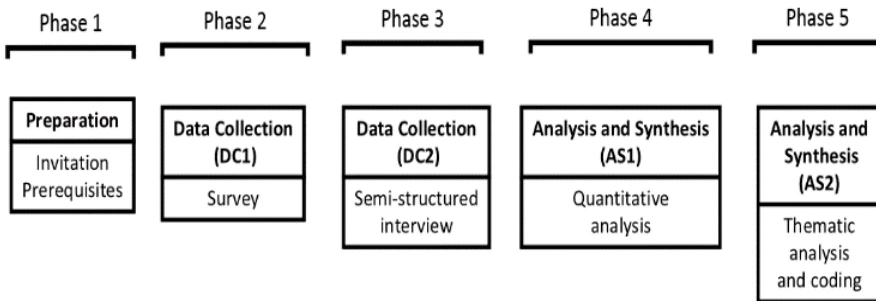


Fig 2, Visualization of the research model and method used in each phase.

18.3.2.1. Preparation

First, the study was presented and discussed during a workshop with software practitioners from software companies within our network, all having an extensive range of software development. Secondly, an invitation to participate in the study was emailed to the participants in the workshop.

18.3.2.2. Data Collection (DC1 and DC2). The data collection in this study was conducted in two phases.

Survey (DC1) - The first data collection phase (DC1) collected data using a survey questionnaire from 17 software practitioners working at four software developing companies. All the practitioners were directly involved in the process of prioritizing TD in their companies' backlog. The survey proceeding was semi-structured, where one of the researchers was available and could clarify and explain the objectives of the survey [256].

المنارة للاستشارات

www.r

As illustrated in Table II, the survey included seven statements assessing the prioritization of TD with respect to the earlier stated research questions. The respondents were asked to indicate their agreement for each statement in relation to their current working situation, using a 6-point Likert scale (not at all, to a small extent, to some extent, to a moderate extent, to a large extent, and to a very large extent).

TABLE II - SURVEYED STATEMENTS

ID	Statement	RQ
ST1	To what extent do you consider your current BM process also includes the prioritization of technical debt?	1
ST2	To what extent do you consider the "gut feeling" having an influence when making prioritization decisions about technical debt?	1
ST3	To what extent do you think that the technical debt's negative effects could be reduced if you did the prioritization of your backlog differently (we are not saying how, just wondering if you think it would be possible or not)	2
ST4	How difficult do you think it is to do the estimation of the value of a refactoring of a technical debt issue?	2
ST5	To what extent do you consider the way you work today with the TD management to be a reactive approach?	1
ST6	To what extent do you consider the way you work today with the TD management to be a proactive approach?	1
ST7	To what extent do you feel that you can influence the decision process of prioritizing the technical debt?	1 and 2

Interviews (DC2) - In the second phase of the data collection (DC2), the survey results were complemented with semi-structured interviews to enrich the collected information and gain a deeper understanding of how TD is prioritized within the BM process.

In total, we conducted focus interviews (as suggested by [73]) with the same 17 practitioners who had earlier participated in the previous data collection phase (DC1). The motivation for selecting focus interviews was based on this methods suitability to uncover factors that influence opinions, behavior, or motivation [146]. The interview questions were developed to cover the same taxonomies as the survey and the identified aspect of our conceptual model as presented in Fig. 1. The interview guide was defined before and used in all interviews. Examples of the interview questions are presented in Table III.

The interviews were conducted between November 2107 and January 2018. The interviews were held either in Swedish or English, and each interview lasted between 90 and 120 minutes. To obtain a more accurate rendition of the interviews, all interviews were digitally recorded and transcribed verbatim. All interviewees were asked for recording permission before starting, and they all agreed to be recorded and to be anonymously quoted for this paper. The interviewer also informed the interviewees that

المنارة للاستشارات

www.r

participation was voluntary and that the interviewee could cancel the interview at any time. Before starting the interviews, the interviewer presented the objective of the study and provided related background information about the research area.

TABLE III - INTERVIEW QUESTIONS

Prioritization Aspect	Examples of Interview Questions
TD Management in Backlogs	• *Briefly describe how your BM works today. What kind of issues are included? One or more backlogs? Who is doing what and when?* • *The strategy for prioritization which you use today, what is the background of it? Used for a long time? Created by whom? Changed recently (or ever)?* • *Do you have a fixed amount of time in each sprint for refactoring of TD today? Pros/Cons with this? How do you know that is enough time, do you do any follow-up on the spent hours?*
Gut Feelings	• *What extent of influence has the "gut feeling" when making prioritization decisions?*
Estimation and Quantification	• *How do you estimate the value from removing TD? What does this gut feeling consist of? Any examples?*
Reactive and Proactiveness	• *Do you consider the way you work today with the TD management to be a reactive or proactive approach?* • *When you do the prioritization of TD today, how long time in advance do you consider (meaning looking at forthcoming features)?* • *Are your roadmap of future features being affected by present TD?*
Prioritization Approaches	• *When prioritizing TD for refactoring, which different factors do you take into consideration and which ones are the most important?*

18.3.2.3. Analysis and Synthesis (AS1 and AS2). The analysis in this study was conducted in two different phases.

Survey (AS1) - The data collected in the surveys during phase 2 (DC1) were analyzed and synthesized quantitatively, i.e., by interpreting the numbers collected from the survey answers. The techniques for analyzing and summarizing the quantitative data include different methods, such as determining measures of central tendency (e.g., median and mode) and measures of variability (e.g., frequencies) [257].

Interviews (AS2) - The analysis and the synthesis of the interview data collected in phase 3 (DC2) were analyzed using thematic analysis [84] to identify, analyze, and report patterns and themes within the data.

The thematic analysis was conducted using a six-phase guide. As a first step, the audio-recorded qualitative data collected from interviews in DC2 were transcribed, and we familiarized ourselves with the data through careful reading of the transcripts. The second step involved the construction of initial codes from the data, where we organized the data into distinct groups. In this phase of the analysis, a qualitative data analysis (QDA) software package called Atlas.ti was used. The third phase focused on identifying themes by sorting the codes into potential themes and collating all the relevant coded data extracts within each theme. Each extract of data was assigned to at least one theme and, in many

المنارة للاستشارات

www.r

cases, to multiple themes. For example, the citation “in that [the backlog], we also enter internal things like refactorings and other requirements that the customer does not have, but the product needs” was coded as “Recorded item in Backlog” in the theme “Backlog Management.” The fourth phase focused on the revised set of candidate themes and the fifth phase focused on identifying the essence of each theme and determining what aspect of the data is captured by each theme. This phase also stressed the importance of not just paraphrasing the content of the data extracts but also identifying what is interesting about them and why. The final phase of the thematic analysis took place when we had a set of fully developed themes; it involved the final analysis and write-up of the publication.

Fig. 3 shows a subset of the outcome of the analysis as they emerged out of the analysis process where the mapping between different code hierarchical levels and the taxonomy are graphically presented. The figure represented a curtailed and reduced part of the data collection model (not completely displayed due to space limitation).

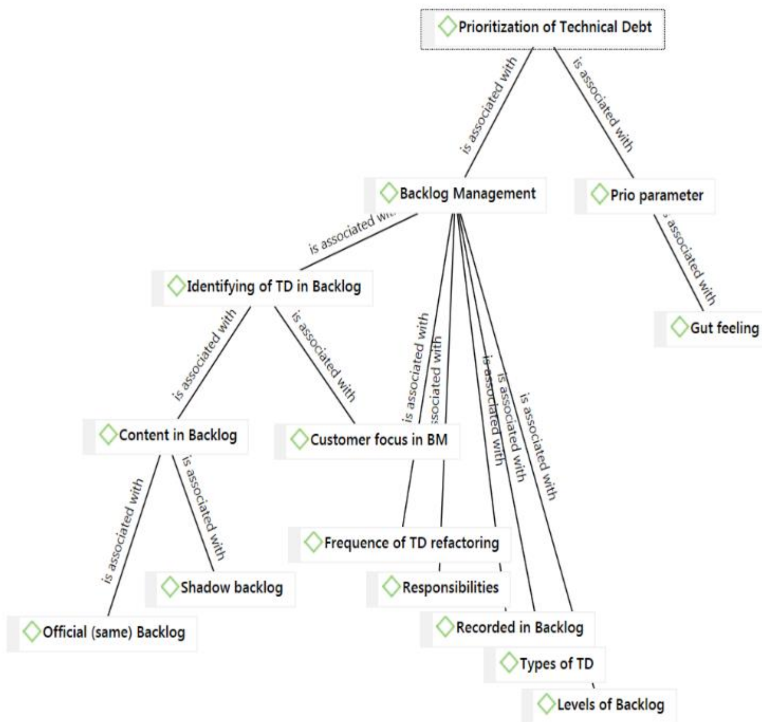


Fig. 3, Sub-set of Coding Scheme

18.4. Results and findings

This section first presents the results from the survey, and then the results from the interviews are presented and organized according to the previously defined research questions.

المنارة للاستشارات

www.r

18.4.1. Survey Results

In the survey, the participants were asked to rate seven sets of 6-point Likert Scale statements (not at all, to a small extent, to some extent, to a moderate extent, to a large extent, and to a very large extent) to indicate how they consider their current prioritization of TD process is carried out. The ratings provided by the respondents for each of the survey statement (see Table II) are presented in Fig. 4.

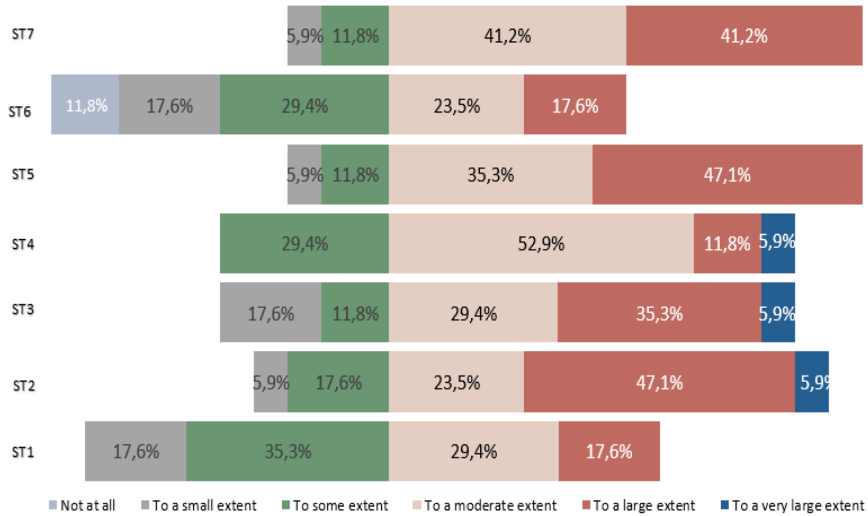


Fig. 4, Summary of the responses to the survey

The first survey result shows (ST1) that all the respondents' BM process included prioritization of TD to some extent, and that three (17.6%) respondents state that their BM process includes the prioritization of TD to a large extent.

According to the survey result (ST2), the gut feeling among the respondents has a profound impact when making decisions related to the prioritization of TD. Thirteen respondents (76.5%) state that gut feeling influences the decision-making of prioritizations related to TD to a large or very large extent.

A substantial part of the survey respondents considers that the amount of TD could be reduced if the prioritization process was carried out differently. According to the result in ST3, all twelve (70.6%) out of seventeen respondents stated that they believed that the negative effects due to TD could be reduced (to a moderate extent, to a large extent, or a very large extent) if they did the prioritization of TD items in their backlog differently.

Survey statement ST4 addresses the extent to which the respondents find the estimation of the prioritization to be difficult. The estimation of the value of a refactoring of a TD issue was considered to be quite difficult. The survey result shows that twelve (70.6%) respondents state that they find the estimation of the value of a refactoring of a TD issue difficult to a moderate, to a large, or a very large extent.

المنارة للاستشارات

www.m

According to the survey result for ST5, the respondents' current TD management approach is mostly reactive, where fourteen (82%) respondents stated that they consider their prioritization process to be reactive to a moderate or a large extent.

On the corresponding statement ST6, assessing to what extent the current prioritization process is considered proactive, seven respondents (41.2%) state that their approach is proactive to a moderate or to a large extent.

Finally, the result of statement ST7 reveals that fourteen (82%) of the respondents feel they can influence the decision process of prioritizing TD to a moderate or a large extent.

18.4.2. RQ1: How is the prioritization of TD carried out and what factors influence the process?

Our analysis has identified many elements that describe how the prioritization process of TD is carried out in practice. We list these elements according to the research model presented in Fig.1.

18.4.2.1. Backlog management and the representation of TD

Deciding which activities should be prioritized and which can be postponed or ignored is one of the most frequently discussed questions during the BM process in software organizations.

All investigated companies used a BM approach when planning the software developing activities. Generally, two types of backlogs were used: the product backlog as an ordered list including everything that is known to be needed in the product and the sprint backlog as the set of backlog items selected to be implemented in the next coming sprint. Meanwhile, the content in the product backlog has a focus on product-level functionality described by user stories; the sprint backlog includes a collection of tasks associated with the user stories from the product backlog.

Typically, the sprint backlog includes a mix of different items (e.g., new features, bugs/defects, and improvements) and is labeled accordingly. Only team B at company B categorized their backlog items using the term technical debt. The rest of the companies categorized TDs by labeling them as "Improvements." Nevertheless, a significant amount of the content of these improvements was considered by the interviewees to be TD.

However, among the different companies and teams, the strategy of whether including TD items in the "official" main sprint backlog or putting them into an unofficial "shadow" backlog differed. Only company B (team A) and company C (team A) included improvement items in their ordinary sprint backlog to some extent. In these teams, only the TD items classified as critical and urgent improvements were added to the ordinary sprint backlog, and the rest of the TD items were managed in a shadow backlog. The usage of shadow backlogs was also used at companies A and D, where these shadow backlogs were commonly not managed together within the ordinary BM management processes. This shadow backlog was described as "A shadow outside as one tries to catch

المنارة للاستشارات

www.r

up and drive aside." This shadow backlog could be managed either on a team or team member individual level (or both).

Several of the interviewees described the reason for using shadow backlogs as an individual decision and a way of being able to handle TD even though the company (or the team) did not support an official representation of TD in the backlogs.

18.4.2.2. The strategy behind the backlog management process

The result indicates that the strategy of managing refactorings of TD in backlogs includes different aspects regarding both the way the TD items existed in the backlogs and how the time, effort, and resources for the refactoring of them were allocated.

Commonly the teams prioritized and selected improvement items from the shadow backlog and introduced them directly into the sprint backlog when they considered the items to be in need of implementation or refactoring, and the team estimated that they had available time for it. However, in some cases, the main sprint backlog was never populated with TD items since the shadow backlog was managed on the side, totally separated from the management of the main backlog. A participant from company B described how they transferred TD items from the shadow backlog into the ordinary backlog: "We also try to add [to the sprint backlog] internal things like refactoring of technical debt and other requirements that the customer does not have, but as the product needs."

Interestingly, the strategy of using shadow backlogs was not described or communicated within any of the companies' ordinary BM process, and the background of using the shadow backlogs was not clear to the respondents. "It is quite interesting [why we have shadow backlogs] because we have always talked about that everything should be included in the [official] backlog. It is very important. We have spread that message throughout the whole organization."

Except for company D, the companies' BM strategy explicitly includes allocating time and resources for refactoring activities. However, this is done in different ways. For instance, at Companies B and C (team C2a), they have on several occasions set aside a full or partial sprint only focusing on improvements from the backlog. At company A, they have a BM strategy allocating 15% of the time in each sprint for improvement work where the developers are specifically encouraged to refactor items from their shadow backlogs. However, this strategy only allows the team to use this amount of time, meaning that there is no obligation to spend the time; furthermore, there is no follow-up on time spent. As one interviewee from company A stated, "This is an old estimation, not sure where it comes from and whether it is enough time."

Company D does not have an explicit strategy for addressing refactoring activities. Their strategy is based on a clustering approach in which the required refactoring activities related to or needed for a specific prioritized feature or bug are refactored.

المنارة للاستشارات

www.r

18.4.2.3. TD prioritization aspects

The companies' shadow backlogs include several TD items which all require refactoring activities. However, these TD items differ in terms of several factors, and these factors need to be considered when prioritizing which item should be selected for refactoring during the next sprint iteration and which refactoring could be postponed.

Commonly, if the TD items were minor in terms of required effort, the team members could prioritize them directly within their teams. Meanwhile, if the TD items were considered to be of a more severe nature, or were estimated to require significant effort, these items were commonly discussed during specific backlog grooming meetings. During these dedicated backlog grooming meetings, none of the companies used a rational and defined prioritization approach to assist the decision-making when prioritizing TD.

The companies described that this step in their BM process was strongly influenced by "gut feelings" among the participants making the decisions. The prioritization process was described to be heavily influenced by the different participants' roles and their empowerment to make decisions during these meetings. For example, one interviewee from company A said, "I think that if we would have the same meeting tomorrow, with the same people, we could end up prioritizing differently, the decision is highly dependent on the discussions that occur in the room, in that specific moment."

Another interviewee at company D described the empowerment of the decision-maker's experience. "In my experience, it's usually the most experienced guy that has the biggest impact [when prioritizing TD]. We don't actually need a big consensus among the participants."

However, even if the prioritization of the TD process was described to a large extent as being influence by meritocracy, several different aspects were mentioned as taken into account when making these decisions, such as the criticality, the urgency, of the TD items where the previous experience and knowledge of the professionals making these decisions had a huge impact on the outcome of the prioritization of TD items.

Our result shows that the decisions based on gut feeling are far from being taken on an ad hoc basis without any firm strategy. It is evident that the different companies focused on different prioritization aspects when doing the prioritization of their TD items. Table IV illustrates the different factors that were described as taken into consideration as a part of the practitioners' gut feelings when prioritizing TD.

المنارة للاستشارات
www.r

TABLE IV - PRIORITIZATION FACTORS – AS PART OF GUT FEELINGS

Current Factors Influencing the TD Prioritization		
Company	***Prioritization Aspect***	***Examples of Argumentation***
A	Risk Assessment	*"We want to fix issues that potentially could make us reach a crisis point. It is also important to focus on the potential risk of actually doing the refactoring."*
B	Product or Business Needs	*"When doing our prioritization, we assess "what is the most important* [TD item to refactor], *either from our* [developmemnt team] *point of view or a business point of view."*
B	Resource Utilization	*"Things gets prioritized because of available resources. We do not want anyone to sit and wait only."*
C	Software Quality	*"I focus a lot on different compromised quality 'ilities,' such as maintainability and flexibility during the prioritization of technical debt."*
C	Financial	*"Management is always interested in ROI. Thus, if you invest money, you should be able to get it back relatively quickly."*
D	Product or Business Needs	*"We do consider future perspectives. About the urgency, the request coming from the surroundings. So, it's really based on experience, on the discussion, on personal feelings."*

Looking at the prioritization aspects in Table IV, it is apparent that there are several different prioritization aspects among the companies, even if these aspects were not explicitly communicated within any of the investigated companies.

18.4.2.4. Pro- and reactiveness

In general, all companies strived toward making proactive decisions during the prioritization process, even if they indicated that, in practice, their decision commonly was taken mostly reactively. Only company B had an overall spoken strategy from upper management stating that all decisions related to the evolution of the software should have a specific proactive focus, where the goal of each decision aimed at surviving in the long term (between 10-20 years) perspective. At the rest of the companies, the prioritization process was described as quite reactive where the TD items were first prioritized when the consequence of them was obvious and caused a direct negative effect. For example, one interviewee shared this, "Our prioritization decisions are not linked to any future solution at all; you just do what you see, reactively, what you see, here and now."

The lack of available information about future product features was commonly expressed as a reason for being more reactive but also other reasons were mentioned by some interviewees: "If you're proactive, you're just referred to as a cost, but if you're reactive, you'll be seen as a guy that fixes things, so you only get the reward if you're reactive

المنارة للاستشارات

www.r

actually". Also, the number of stakeholders and users operating the software described having an influence on the reactiveness and the proactiveness: "Some time ago when we didn't have a lot of users, then I think we were more proactive in what we did. And now when we have more users, we are more reactive in that sense."

Although most of the interviewees described their prioritization process by not taking the evolution of the software into consideration, a proactive approach of the prioritization process of TD was described as a highly desired approach. One interviewee at company C said, "You want to be proactive, but you do not often have all the data and information available [about the future of the software], so it's often popping up things, and first then you can act."

18.4.3. RQ2: What Are the Challenges of Prioritization TD?

In this section, we explore how software companies' reason about the challenges associated with the prioritization process of TD. In general, the prioritization process of TD was rated high among the challenges related to the overall prioritization process. While analyzing the identified challenges, it became clear that several of them were related and, therefore, we have grouped them accordingly.

18.4.3.1. Predicting the future of the software

One general major concern for the interviewees was that they lacked information about how the software was going to evolve in the future, and thus they found it difficult to prioritize necessary and urgent TD items for refactoring. Yet another challenge related to this area refers to the challenge of motivating the prioritization of TD without a clear picture of the future of the software due to uncertainty. The following quote from company C describes this challenge: "I'm always saying that, in this project, we may use parts of this software in other projects in the future as well. But then I get questions like "Yes, but it will never happen. Why do you say that? Why should we put this cost now for governance, it will never happen." No, but my gut feeling says we should do it. However, if we had figures saying that there is a 20% chance that we will reuse part of this software in the future, that would have helped us making those prioritization decisions more accurate."

18.4.3.2. Available information about the TD items

The most critical limitation for the prioritization of the TD process was expressed in terms of the lack of available information about the TD items and its related negative consequences, both from a current perspective but also from a future perspective. One interviewee said, "I would like to integrate both the costs of TD and probability [of the occurrence of the TD interest payment] and use more business decisions….Both costs from a current status perspective but also from a future cost perspective".

المنارة للاستشارات
www.r

Several interviewees stated that the lack of information was the missing key to understanding the value of refactoring of TD and thereby being able to reduce the negative consequences of TD. Having access to reliable and updated estimates and/or quantifications of the negative effects of the present TD items was described as crucial for being able to improve the process of prioritizing TD. However, none of the companies had access to or were able to produce this kind of information for the identified TD items. Neither did any of the companies use any supporting software tools to assist in the decision-making process of prioritizing TD in their backlog.

18.4.3.3. TD refactoring competition with customer requirements

Our findings indicate that the pressure of delivering customer value and meeting delivery deadlines forces the software teams to down-prioritize TD refactorings continuously in favor of implementing new features rapidly.

Furthermore, our findings also indicate that this pressure depends largely on whether the user of the software is an internal or external customer. If the user is an internal customer, the pressure is much lower than if the user is an external customer. A lower pressure induces TD refactorings to receive more attention and thus be more easily prioritized. The teams at company C exemplify this situation, where the two teams have different types of customers. As one of the team members put it, "It is the opposite situation for us in our team [having an internal customer]. They [the other team, having an external customer] work significantly more with a customer focus in their backlog. We have a little more free stuff, and therefore, we can prioritize specifically technical debt issues. We don't suffer from such time pressure, and we can also control our documentation better, and are more flexible, and we ourselves are also able to influence to a larger extent the decisions that are made".

18.5. Discussion

Through this study, we have identified several challenges software practitioners face when prioritizing TD within their BM process. We also examined what different BM strategies were used and what factors influence the decision-making process of prioritizing TD.

When TD is present in the software, the only significantly effective way of reducing it is to refactor it. However, to be refactored, the refactoring activities of the identified TD items needs to be prioritized in competition with, for example, implementation of new features. From a software practitioners' point of view, the process of prioritizing TD is surrounded by several ambiguities and difficulties.

Firstly, a starting point of our findings shows that TD items commonly are not present in the main sprint backlog; they are often documented in quite ad-hoc managed shadow backlogs. Even if the TD items sometimes are escalated by transition from the shadow backlog into the main backlog, this transition is not clear in terms of when and if the TD item should be transferred. This way of administrating TD items could potentially lead to the TD items becoming not fully visible and known during the prioritization process, and

المنارة للاستشارات
www.r

thereby, not given sufficient attention. There is also a risk that the professionals prioritizing the work for each sprint are not aware of the present TD items (since they are locally managed in the teams), and therefore, these items will never be considered during the prioritization process.

Secondly, the TD prioritization decision-making process has earlier been studied by other researchers, but until now, not from a concrete and practical perspective. Our findings show that when taking these prioritizations of TD decisions, the practitioners rarely base their decisions on pre-defined procedures or guidelines. Rarely do they conduct any cost-benefit analysis or calculate/estimate TD as an investment or calculate/estimate the current or future interest rate in terms of maintenance costs. Neither do they carry out any severity analysis. A possible explanation for these results may be the lack of adequate information about the TD items to assist such activities and a lack of delegated responsibility for providing such information coupled with upper management not focusing strategically and explicitly on remediation of TD in the software. Quite the contrary, our findings show that the decisions related to prioritization of TD are heavily influenced by gut feelings and on meritocracy where the experience of the decision-makers have a huge impact on the outcome.

Further, our findings show that the prioritization process is also highly dependent on individual practitioners' power to justify and argue for a certain item in the backlog to be prioritized. This leads to polarization of the TD prioritization process because it is often perceived as selecting one person's opinion over another's. Further, our result indicates that decisions related to prioritization of TD often are not backed up with validation. This result may be explained by the fact that the companies lack sufficient guidelines and supporting frameworks guiding the prioritization process of TD and directing the discussion into specific areas of interest.

Thirdly, our results show that refactoring activities of TD get less attention if the software will serve an external customer, compared to an internal customer since, in these cases, the focus on delivering customer value is being prioritized in favor of refactoring activities of TD. This result may be explained by the fact that management is not fully aware of the magnitude of the consequences having TD in the software can cause, both in terms of, for example, compromised software quality, lowered developer productivity, increased maintenance cost, and project delays [144].

The upper section of the visualization in Fig. 5 shows contributing causes of why the prioritization of TD is down prioritized or neglected and the lower section presents the identified related challenges for the prioritization of TD in backlogs. This figure can assist in developing actions that sustain the enhancement of the TD prioritization process.

The figure illustrated that there are several identified causes that potentially have an impact on the TD prioritization process. In the future, we intend to continue with the validation and refinement of the causes by applying them to additional industrial cases and investigating the validity of the findings.

المنارة للاستشارات

www.r

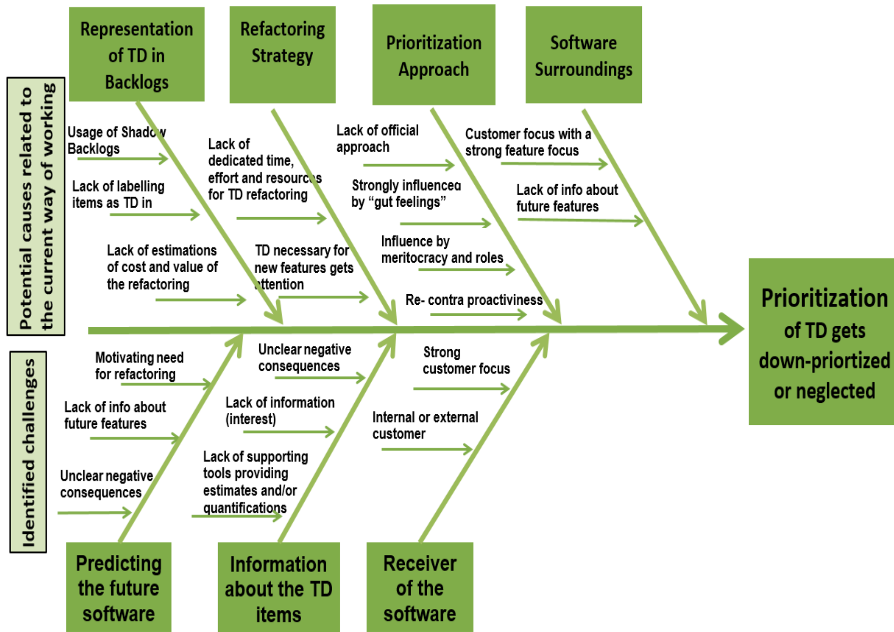


Fig. 5. Identified Causes and Challenges Related the Prioritization Process of TD in Backlogs

18.5.1. Implications for practitioners and researchers

When looking at the process of prioritizing TD items during the BM process from a practitioner's perspective, we conclude with some important implications and recommendations for practitioners and researchers, particularly useful during prioritization of the TD in the BM process:

- To give the refactoring of TD activities adequate attention, it is crucial that the professionals prioritizing features and improvements (e.g., TD items) in the backlog are aware of the magnitude of the negative consequences TD brings to the software, both from a short-and a long-term perspective.
- The use of ad hoc managed shadow backlogs leads to the need to define if and when this TD item should be prioritized and whether shadow backlogs should be transferred to the official sprint backlog. Otherwise, there is a risk of missing or down-prioritizing these TD items.
- It is necessary to agree and communicate a shared vision and purpose of the product and its features to all practitioners who are involved in prioritizing TD (both using a shadow backlog and the official sprint backlog). With such proactive prioritization approach, aligning the product roadmap to specific prioritization tasks will make the decision-making more transparent and understandable.
- A guiding framework or approach in which different prioritization aspects are included can support the grooming process of identifying TD issues that are important to prioritize based on different goals and deliverables. The use of such a framework or approach would also contribute to traceability enhancement.

المنارة للاستشارات
www.m

18.6. Study Limitations

It is important to consider the validity of the results in a case study [73]. The findings in this study are subject to at least three different limitations.

First, the limited number of participating companies and the qualitative nature of the investigation means that our findings need to be interpreted cautiously. Although the findings cannot be generalizable to all software companies, they provide good evidence about the identified five dimensions on which the research questions were based, and the findings also point to areas of further research. Furthermore, we plan to expand our sample in the future to reach a higher degree of validation of our results.

Secondly and correspondingly, the goal of collecting quantitative data using a survey was not meant to give precise, measurable results but rather to indicate the way the practitioners perform their prioritization of TD today.

Thirdly, the selection of the companies was based on an available convenience sample among companies within our network that was available and had a specific interest in participating in this study. However, the participating companies represented all software companies from different business areas and with a software development process including regular BM prioritization activities.

Taken together, these are all limitations that can be considered acceptable in light of the exploratory purpose of this study. We preferred to gain a deep and rich understanding of the context of a few cases to build a holistic first theory rather than surveying the topic on a high level only.

18.7. Conclusion

This study set out with the aim of assessing how the prioritization of TD is carried out in practice by practitioners in today's software industry. The results of this investigation show that the prioritization of TD in backlogs is commonly done in an unstructured way, where the identified TD items are often managed in "shadow backlogs," potentially resulting in the TD items being overlooked during the prioritization process.

Other findings show that the process regarding which TD item to prioritize is heavily influenced by gut feeling among the decision-makers, and whether the user of the software is an external or internal character also influences the prioritization strategy of TD. Several different challenges for prioritization of TD were revealed in this study, where one of the more significant challenges refers to the difficulty of estimating the value of each potential TD refactoring activity. Further on, even if the software practitioners have a vision of acting proactively when prioritizing TD while the circumstances and the organizational culture force them to act reactively upon the prioritization tasks. Overall, the study indicates that TD needs further attention in the overall BM process, and the findings can also assist in explaining why TD commonly is not sufficiently visible in the official backlogs and thereby not prioritized accordingly.

المنارة للاستشارات

www.r

19. Carrot and Stick approaches when managing Technical Debt

This chapter aims to explore how software companies encourage and reward practitioners for actively keeping the level of technical debt down and also whether the companies use any forcing or penalizing initiatives when managing technical debt.

When developing software, it is vitally important to keep the level of technical debt down since it is well established from several studies that technical debt can, e.g., lower the development productivity, decrease the developers' morale, and compromise the overall quality of the software. However, even if researchers and practitioners working in today's software development industry are quite familiar with the concept of technical debt and its related negative consequences, there has been no empirical research focusing specifically on how software managers actively communicate and manage the need to keep the level of technical debt as low as possible.

This study aims to explore how software companies encourage and reward practitioners for actively keeping the level of technical debt down and also whether the companies use any forcing or penalizing initiatives when managing technical debt.

This paper reports the results of both an online web-survey provided quantitative data from 258 participants and follow-up interviews with 32 industrial software practitioners. The findings show that having a TD management strategy can significantly impact the amount of TD in the software. When surveying how commonly used different TD management strategies are, we found that only the encouraging strategy is, to some extent, adopted in today's' software industry. This study also provides a model describing the four assessed strategies by presenting its strategies and tactics, together with recommendations on how they could be operationalized in today's software companies.

This chapter has been published as:
Carrot and Stick approaches when managing Technical Debt
T. Besker, A. Martini, and J. Bosch
Third International Conference on Technical Debt, co-located with ICSE, South Korea, 2020.
This paper was grated the *Best Paper Award.*

19.1. Introduction

When developing software, it is vitally important to keep the level of technical debt (TD) down, since it is well established from several previous studies that TD can, for example, lower the development productivity [7], decrease the developers' morale [187], and compromise the overall software quality [15] and even lead to a crisis point when a huge, costly refactoring or a replacement of the whole software needs to be undertaken [258].

The TD metaphor was first introduced by Ward Cunningham [186] to illustrate the need to recognize the potential long-term negative effects of immature code that is sub-

www.r

optimally implemented during the software development lifecycle. This debt must be repaid with interest in the long term [188].

Even if the concept of TD and its negative consequences are quite well known to software engineering (SE) practitioners today, there is always a risk that TD remediation tasks gets down-prioritized or neglected by the practitioners, since today's' software practitioners face increased pressure from management to reduce the development time and thereby to reduce the costs of the development [259]. On the other hand, it is, at the same time, important to deliver high-quality software with as little TD as possible. This balancing act between implementing and delivering the software as fast as possible and at the same time spending time and effort on both avoiding introducing TD in the first place and conducting TD refactoring activities of already implemented software gets particularly demanding. Previous studies have shown that it is quite challenging for software practitioners to make decisions themselves on how to prioritize the time and effort they should spend on TD remediation tasks [259].

Similar to other professionals are the software engineers' work outcome, their attitudes, and their work behaviors, influenced by the company's corporate culture together with the mindset from the managers. This means that managers can have an outsized impact on the overall software development process by adopting different management strategies and using techniques for controlling and directing the software engineers to achieve predetermined goals.

In recent years, the use of different strategies in behavioral interventions has become more popular [260]. In a literature review, this study initially identifies four different strategies that managers can adopt to impact how practitioners work with TD. Besides encouraging employees by, for example, introducing training programs that focus on raising awareness and enhancing knowledge about specific desired behavior, there are also other strategies managers can implement to impact their employees [57]. In general (in several different roles and fields), one mechanism managers use to impact practitioners' work is an incentive program, where a specific behavior is recognized and rewarded [261],[189]. Oppositely, managers can also use disincentive programs to penalize an undesired or destructive behavior [208]. Further, managers can similarly implement explicitly forcing requirements and rules, where all concerned employees must fulfill and adapt to in order to continue their work or, for example, to deploy their implementations and continue developing new tasks [262].

However, to the best of our knowledge, this is the first study to investigate empirically how common and important different management strategies are when specifically managing TD in today's software development industry. Additionally, this study also surveys how practitioners in software companies perceive such strategies. In particular, this study examines four research questions.

RQ1: How common is an *encouraging* attitude to keep the level of TD down, and do software engineering practitioners perceive this TD management strategy as an effective or desirable strategy?

المنارة للاستشارات

www.m

RQ2: How common are *rewarding* incentives to keep the level of TD down, and do software engineering practitioners perceive this TD management strategy as an effective or desirable strategy?

RQ3: How common is it to use a *forcing* mechanism to keep the level of TD down, and do software engineering practitioners perceive this TD management strategy as an effective or desirable strategy?

RQ4: How common are *penalizing* disincentives to keep the level of TD down, and do software engineering practitioners perceive this TD management strategy as an effective or desirable strategy?

In previous research, most of the attention to TD has focused on the usage of different tools to manage and keep TD at bay, while none have investigated TD from the managers' perspectives where the important context of adopting different TD managing strategies has been in focus.

This paper reports the results of both an online survey providing quantitative data from 258 respondents and qualitative data from interviews with 32 software practitioners from seven software companies.

The contribution of this paper is fourfold. First, we show how commonly used each of the investigated strategies are within today's software industry. Second, we present a TD management quadrant model that presents four different TD management strategies and illustrates its strategies and tactics, together with recommendations on how to implement them. Third, our result shows that a TD management strategy can significantly impact the amount of TD in the software. Fourth, when surveying how commonly used different TD management strategies are, we found that only the encouraging strategy is, to some extent, adopted in today's' software industry. Taken together, these findings provide valuable insights into the role management has on the way practitioners address TD during their software development work.

The rest of this paper is organized as follows. Section 2 presents the used conceptual framework, and Section 3 introduces related work. Section 4 describes the research methods in detail. Section 5 presents the research results. Sections 6 discusses the findings, and Section 7 presents Implications for Future Practice. Section 8 describes threats to the validity of the study, and Section 9 concludes the study.

19.2. Conceptual framework

As briefly described in Section 1, there are different management strategies that organizations can use to influence employees' working behaviors. Initially, we performed a literature review by searching for research publications addressing different strategies that managers can use for influencing practitioners' working behaviors and attitudes. Based on the outcome of this review, we have depicted a conceptual model, as presented in Fig.1 and formulated the research questions based on this framework. The framework illustrates four different management strategies;

المنارة للاستشارات

www.r

a)*Encouraging*, b) *Rewarding*, c) *Forcing*, and d) *Penalizing*, where each of them connects to one of the research questions presented in Section 1. The conceptual framework with its four main strategies is proposed under the rationale that these four strategies potentially are important and can have an influence on the reduction of TD and potentially are being used by managers to manage the amount of TD during the software development work.

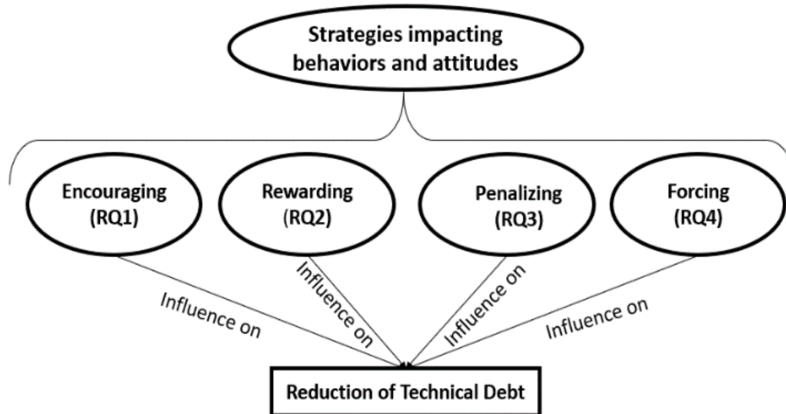


Figure 1: Conceptual framework

19.3. Background and related work

This section presents related work concerning incentive and disincentive programs in today's software engineering field, followed by the different management strategies, as illustrated in the conceptual framework in Fig. 1.

19.3.1. Incentives and disincentive programs in SE

An incentive program is a program addressing a planned activity designed to motivate employees (individuals or teams) to achieve specified and predetermined organizational goals or objectives within a specific time frame. Whereas a disincentive program is the antonym of the incentive program and discourages employees (individuals or teams) from performing specific activities [263].

Commonly, software development projects are measured using financial indicators where, typically, the goal of the incentive program is to give bonuses to managers who run their projects with a high margin of profit or within the budget timeframe [264]. There is no research to date on how other software engineering roles, such as, for example, developers, testers, and architects, are included in incentive or disincentive programs in general and, more specifically, TD management.

المنارة للاستشارات

www.r

19.3.2. Encouraging activities

Encouraging employees is an important part of being a leader where the leader highlights and compliments specific desired actions and where the leader also provides constructive criticism if needed. Managers' behaviors provide an important message to employees, meaning, for example, that a high level of creativity and innovation result from managerial behaviors [265] where the relationship between employees and their managers has a significant bearing on employees' work-related attitudes and behaviors [57]. By default, just encouraging employees does not include any direct rewards.

19.3.3. Rewarding incentive

Several studies show that reward and recognition programs can positively influence motivation, performance, and interest within an organization [261],[189]. The overall goal with reward and/or recognition programs is to foster teamwork, boost employee loyalty, and ultimately facilitate the development of a wanted culture that rewards a specific behavior [261]. The practitioners (individuals or teams) who fulfill the goals get a predefined reward. A reward program can, for example, recognize developers who adopt suggested techniques, and thereby the reward incentive gives a significant boost to those who deploy best practices, where the achievement, for example, can be rewarded by a badge [266] or by a gift card or a monetary reward.

19.3.4. Forcing mechanisms

The strategy based on forcing mechanisms refers to mandatory rules and requirements that need to be fulfilled and follow by the practitioners in order to demonstrate adherence to methodologies, rules, regulations, guidelines, or best practices [262]. This could be exemplified by a situation in which mandatory rules and requirements are not met and hence forcing the practitioners to go back and alter the software before being allowed to continue with, for example, adding additional features or deploying the software. Examples of commonly adopted rules and requirements from an SE perspective are not allowing any bugs in the software and requiring that the software be fully tested before deployment and the code fully reviewed and that it follows the coding standards.

19.3.5. Penalizing disincentive

Organizational penalty or punishment) is a pervasive phenomenon in many companies and organizations [267] that yields penalties for an undesired behavior and are also referred to as a disincentive strategy. Penalization refers to when managers apply a negative consequence or the removal of a positive consequence following an employee's undesirable behavior, intending to decrease the frequency of that behavior [208]. According to Wang and Zhang [267], some software development organizations have adopted punishment measures in an attempt to improve software developers' performance, reduce the software defects, and hence ensure software quality. Their result

المنارة للاستشارات

www.r

shows that while a penalty mechanism helps to reduce software defects in daily coding activity, it fails to achieve programmers' maximum work potential. In their study [267], penalty rules were introduced when software developers were tracked submitting unsuccessful submissions, which caused monetary fines for the individual developer.

Since there is a current gap in research addressing how different TD management strategies impact how practitioners work with TD, this study built on research conducted in other domains and examines the current state of different management strategies in the SE field. Our work is, therefore, different from above-mentioned studies in several aspects: We a) provide results derived from data from a real software development environment rather than discussion without empirical evidence as support, b) we combine both qualitative and quantitative methods, c) our study investigates four different management strategies, and d) our investigation primarily focuses on TD.

19.4. Methodology

As visualized in Fig.2, this study used a combination of quantitative and qualitative research approaches, and the research design was divided into six phases. The following sections describe each phase and its related research methods.

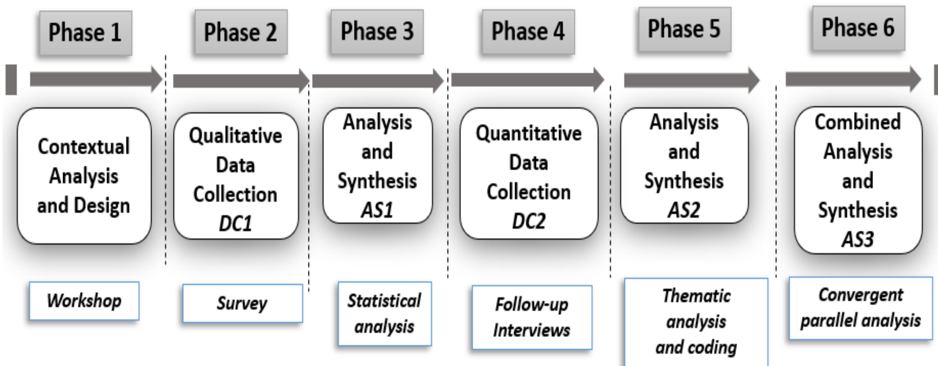


Figure 2: Research Design

19.4.1. Phase 1 – Preparation and Design

The study was first presented and discussed during a workshop with software practitioners from seven software companies within our industrial software network. All companies had an extensive range of software development. The goal of this phase was to guide how to collect information regarding the studied strategies and to select the most suitable research model to use. The outcome of this phase is the research model shown in Fig. 1, which directed the design, data collection, and analysis of the following phases.

This study's selection of participating companies was carried out with a representative convenience sample [86] of software professionals from seven of our industrial software developing partners.

المنارة للاستشارات

www.r

19.4.2. Phase 2 – Data Collection (DS1)

The data collected in this phase was supported by an online web survey, which was designed and hosted by the survey service SurveyMonkey.

The motivation for using a survey in this part of the study was to reach a high level of generalizability based on a large population of software professionals and to achieve an understanding of that population [268].

According to the guidance provided by [77], the first draft of the survey was tested by four industrial practitioners (developer, manager, project owner, and software architect) and by two Ph.D. candidates in order to evaluate the understanding of the questions and the usage of common terms and expressions [77]. During this evaluation, we also monitored the time needed to complete the survey.

The survey invitations were emailed directly to seven companies within our networks, all located in Scandinavia, having an extensive range of software development, and invitations were also published on software engineering-related networks on LinkedIn. After two weeks, a reminder was sent out to those who had been specifically invited. The surveys were anonymous, and participation in the surveys was voluntary. Across all these collaborators, 312 respondents began the survey, and 258 respondents answered all questions. Due to high completion rates (~83%), we decided to reject the incomplete responses, according to the guidelines proposed by Kitchenham and Pfleeger [78].

The first part of the survey gathered descriptive statistics to summarize the backgrounds of the respondents and their companies. These data are collected and presented in Table I.

The respondents' level of education was relatively high, with 58% with a master's degree and 25% with a bachelor's degree. The survey included respondents having different roles, where 49% were developers/programmers/software engineers, while 25% were software architects. Approximately 78% of the software architects and 59% of the developers/programmers/software engineers had more than ten years of experience. The most common size of the software development team was 6–10 members (36%), and most (32%) systems were, on average, 5–10 years old from their initial design. Nevertheless, most of the software systems were embedded systems (49%) and real-time systems (42%). However, in this specific survey question, the respondents could select more than one option.

The second part of the survey included the four survey statements (ST) presented in Table 2, to facilitate quantitative answering the research questions presented in Section 1 (when fully answering the RQs, we used quantitative data from this phase in combination with qualitative data in next coming phase in section 4.4 as described in section 4.6).

For each of the statements, the respondents were asked to indicate their level of agreement on the 6-point Likert scale; Strongly Agree, Agree, Somewhat Agree, Somewhat Disagree, Disagree, and Strongly Disagree

المنارة للاستشارات

www.r

TABLE 1: CHARACTERISTICS OF THE SAMPLE SURVEY

Factor	Percentage split	
Experience	< 2 years 2 - 5 year 5 - 10 year > 10 years	3,90% 10,50% 17,40% 68,20%
Education	Master's degree Bachelor's degree No Univ. education Other: PhD degree	57,80% 25,20% 6,20% 5,80% 5,00%
Roles	Developer/Program/Software Engineer Software Architect Manager Project Manager Product Manager Expert Tester	*49,20% 24,80% 6,20% 6,20% 5,0% 5,0% 3,50%
Gender	Male Female Other/no share	89,90% 8,50% 1,60%
Team size	1–5 members 6–10 members 11–20 members 21–40 members > 40 members	23,30% 36,00% 15,90% 6,60% 18,20%
Software system type*	Embedded system Real-time system Data management system System integration Modeling and/or simul. Data analysis system Web 2.0 / SaaS system Other	48.84% 41.86% 22.09% 20.93% 15.12% 14.73% 8.53% 8.53%

* More than one option was selectable

المنارة للاستشارات

www.r

TABLE 2: CHARACTERISTICS OF THE SAMPLE SURVEY

ID	Statement	RQ
ST1	Our team or I am explicitly rewarded if TD is kept down.	2
ST2	Our team or I am explicitly penalized if TD is not kept down.	4
ST3	Our team or I am explicitly forced to keep the level of TD down (i.e., to be allowed for deployment)	3
ST4	Our team or I am explicitly encouraged if TD is kept down.	1

19.4.3. Phase 3 – Analysis and Synthesis (AS1)

The data from the survey were analyzed quantitatively, that is, by interpreting the numbers obtained from the answers. The data were analyzed using descriptive statistics and graphically visualized in a diverging stacked bar chart.

19.4.4. Phase 4 – Data Collection (DS2)

In this stage, the second round of data collection was conducted, where we group-interviewed 32 software practitioners. Interviews can be divided into unstructured, semi-structured, and fully structured interviews. As suggested in [73], this study employed the technique of semi-structured interviews, where the questions were planned but not necessarily asked in the same order as they were listed.

This semi-structured interview technique allowed flexibility and exploration of the studied objects by asking follow-up questions based on the respondents' answers in the previous survey (as described in 4.2), meaning that these interviews were used for obtaining detailed information about the interviewees' perceptions and interpretations of the study topics.

All interviews were focus-group interviews based on guidelines by Krueger and Casey [146], stating that this method is specifically suitable, serving as a source of follow-up data to assist a prior used data collection method: "*The researchers need the information to help shed light on quantitative data already collected*."

In total, we interviewed seven companies, where each interview included between four to seven participants. These companies all participated in the previous survey, and all the interviewees had answered the survey before the interview. Altogether, we interviewed 32 experienced software development professionals, with roles as architects, developers, product owners, and managers. For confidentiality, interviewees and their companies are kept anonymous.

All interviewees were asked for recording permission before starting, and they all agreed to be recorded and to be anonymously quoted for this paper. Each interview lasted between 105 and 120 minutes and was digitally recorded and transcribed verbatim.

المنارة للاستشارات
www.r

Before the interviews started, the compiled results from the previous survey were presented to the respondents (using graphical illustrations such as bar diagrams and graphs. This presentation allowed the respondents to relate the interview questions more easily to the results of the survey.

The interview questions were designed to a) increase the understanding of the survey results, b) ensure that the questions in the survey were understood and interpreted as intended and uniformly, c) confirm the results of the survey, and d) understand the implications of the survey results. The questions were developed to cover the same taxonomies as the previous survey to validate the findings of the survey. The validation was performed by asking the interviewees if they agreed (or not) to the findings by comparing the previous quantitative results with their experiences in order to understand if the findings confirmed or contradicted each other. The interview guide was defined before and used in all interviews. Examples of interview questions for each RQ are presented in Table 3.

19.4.5. Phase 5 – Analysis and Synthesis (AS2)

This stage focused on analyzing the data collected in the previous Phase 4 (Section 4.4). The analysis and the synthesis of the data were analyzed using thematic analysis [214].

First, the transcriptions from the recorded interviews were manually coded using established guidelines in the literature [147]. The tracking of the links between the codes and the quotations was supported by a Qualitative Data Analysis tool called Atlas.ti.

Secondly, based on the research taxonomy, a coding scheme containing four broad categories and 19 individual codes was developed. Fig. 3 shows the outcomes of the analysis process, where the mapping between the different hierarchical categories (highlighted in grey) and individual codes is presented graphically (the arrows between the entities in the figure show the relationship between the codes and the sub-codes). For example, the citation “I think it sounds like the reward would be the best way of keeping the measure of technical debt down” was coded as “Rewarding.”

To ensure that the coding was performed consistently and reliably, two authors of this study synchronized the output of the codings, as suggested by Campbell et al. [85].

المنارة للاستشارات
www.m

TABLE 3: EXAMPLES OF INTERVIEW QUESTIONS

TD Management Strategy	Examples of Interview Questions
Encouragement (RQ1)	• How do you perceive encouragement from managers for keeping the level of TD down? • How could such a strategy be implemented? • Do you agree with the results of the survey?
Rewarding incentive (RQ2)	• How do you perceive a rewarding incentive for keeping the level of TD down? • Do you have this or a similar strategy in place or planned for the future? • How could such a strategy be implemented?
Forcing (RQ3)	• How do you perceive a forcing mechanism for keeping the level of TD down? • Do you have this or a similar strategy in place or planned for the future?
Penalizing disincentive (RQ4)	• How do you perceive a penalizing disincentive for keeping the level of TD down? • How could such a strategy be implemented? • Does your company have a team and/or personal incentive/disincentive systems for any other kind of quality criteria related to your software development process?
Other	• Which strategy of keeping the TD down, do you consider to be the most/least successful (and why) and under what circumstances?

19.4.6. Phase 6 – Combined Analysis and Synthesis (AS3)

In the sixth phase of the study, we combined the results we received in the previous quantitative and qualitative data collection and analysis stages. When interpreting the results from a mixed-methods research approach, there are different designs to facilitate providing a stronger interpretation and greater insight from the results.

This study has used the convergent parallel mixed-methods design when collecting both qualitative and quantitative data, analyzed them separately, and compared the results to understand whether the findings confirmed or contradicted each other [66]. The goal of using this approach was to strengthen, and thus confirm our findings after the results had been generated from the data [170].

المنارة للاستشارات

www.r

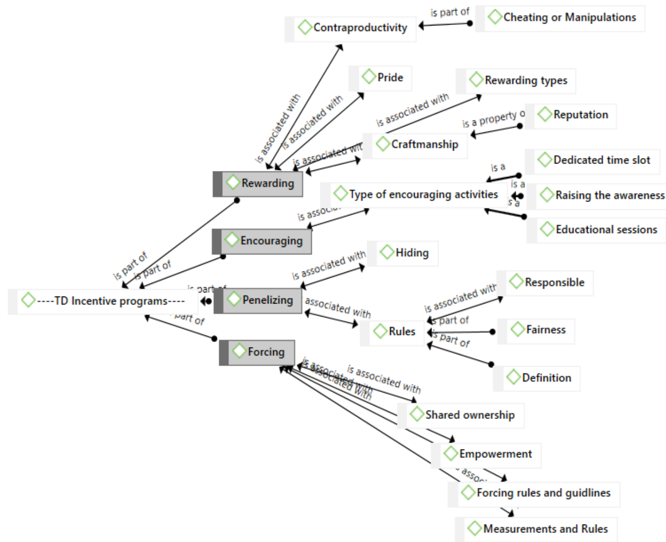


Figure 3: Coding Scheme

19.5. Results and findings

In the survey, the participants were asked to rate their agreement with four statements (ST1-ST4) using a 6-point Likert Scale. The ratings provided by the respondents for each of the survey statements (as presented in Table 2) are presented in Fig. 4, and further described in the following subsections.

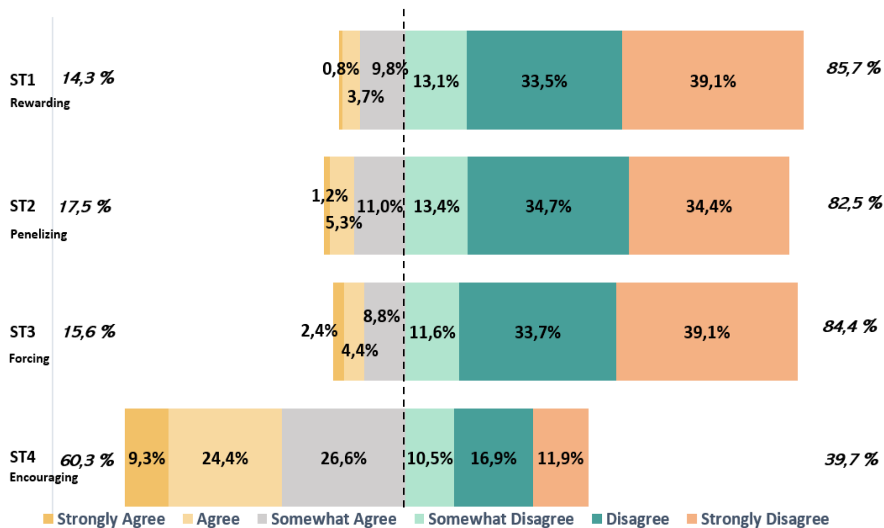


Figure 4: Summary of the responses to the survey

المنارة للاستشارات
www.r

19.5.1. Encouraging strategy

The aim of the first research question (RQ1) is to investigate how common an encouraging strategy is to keep the level of TD down and how software engineering practitioners perceive this TD management strategy.

19.5.1.1. Survey results.

When looking at the results from the different statements in Fig. 4, it is evident that one of the statements excels compared to the other statements.

What stands out in the figure is the statement assessing whether the respondents are encouraged to keep the level of TD down (ST4), where 60.3% of the respondents agree to some extent of being encouraged, and 39.7% disagree to some extent of being encouraged to keep the level of TD down.

A remarkable result of this statement is that 11.9% of the respondents strongly disagree with being encouraged to keep the level of TD down.

19.5.1.2. Effective or desirable strategies.

The attitudes among the practitioners toward introducing TD or conductin TD remediation tasks were described as being guided and targeted by the mindset and the attitudes from the management side where recognition of leaders and peers were important components. This meant that when management was focusing its attention on the importance of TD, the employee was encouraged to focus his or her work in the same direction.

All the interviewees considered an encouraging managing strategy of TD as highly effective and impactful, and several of them described that this strategy clearly could be more emphasized within their organizations and thereby also have a more significant impact on the amount of TD in their software.

19.5.1.3. Tactics for encouragement.

Several actions were identified as examples of encouraging activities from management to keep the level of TD down. Several of the interviewed companies strived to continuously raise awareness about the concept of TD and its related negative consequences as an encouragement to keep the level of TD down. Some companies ran satisfyingly dedicated educational sessions to explicitly address how to avoid the introduction of TD in the first place. One interviewed software architect said, “We [the architectural team] try to help some teams here that we are close to, and we try to teach them how to write good APIs.” Yet another interviewee from another company echoed this view, “I think of education. I think a lot of debt is introduced because of a lack of knowledge.”

The managers for some teams in one of the interviewed companies had set aside a specific amount of working time within each sprint to allow explicitly (without imposing on the

المنارة للاستشارات

www.r

developers) spending time on TD remediation activities together with other software-improving activities. This dedicated time slot encouraged the involved engineers (such as testers, developers, and architects) to focus on TD issues in every sprint as an incorporated part of their overall working process.

19.5.2. Rewarding Incentives

The second research question (RQ2) addresses how common rewarding incentives are to keep the level of TD down and how software engineering practitioners perceive this TD management strategy.

19.5.2.1. Survey results.

As illustrated in the first statement (ST1) in Fig. 4, only 14.3% of the respondents agree to some extent with being explicitly rewarded when keeping the level of TD down, and thus 87.7% of the respondents state that they are not explicitly rewarded for it. What stands out in this data is that only two (2) respondents state that they strongly agree with being rewarded when keeping the level of TD down; meanwhile, a hundred (100) respondents state that they strongly disagree with being explicitly rewarded if they keep the level of TD down.

19.5.2.2. Effective or desirable strategies.

None of the interviewed companies had an incentive program where employees were rewarded for any (not only TD specifically) explicit behavior. The thoughts among the interviewees differed as to whether adopting rewarding incentives is an effective or desirable strategy to keep the level of TD down.

Some interviewees were skeptical of explicitly highlighting and rewarding specific working activities and behaviors such as TD remediations since they thought keeping the level of TD down should be an activity that comes with the craftsmanship of software development and the working pride of software engineers. For example, one interviewee said, "It sends the wrong signals if you start to reward or penalize individuals or teams for something that should be a normal part of their daily work. Because I see that paying off technical debt should be done as a part of your daily work to the extent that is possible".

On the other side, several other interviewees argued that a TD managing strategy based on rewards could be effective since rewards can motivate practitioners to manage TD further. One such interviewee said, "I guess it is a good way. If you have this kind of reward now and then, you keep up the awareness that this is important. It is something of a signal from the organization."

المنارة للاستشارات

www.r

19.5.2.3. Tactics for rewards.

Concerns regarding the different appropriate types of rewards were widespread. Some proposed (since none of them hade any incentive programs in place) rewards such as monetary compensation, extra holidays, and pizzas to the teams. Meanwhile, other interviewees said a reward does not have to be tangible; it could be a simple acknowledgment since official praise and thus an enhanced reputation are considered equally important: “A reward does not have to be money. You get some reputation that will actually be enough reward in itself.”

Nevertheless, even if a rewarding incentive has the best intention to decrease the amount of TD in the software, such initiatives could easily be misused by causing a counterproductive backlash where, for example, practitioners primarily focusing on TD remediation tasks in order to get the rewards or focusing only on the TD items that are easy to refactor, and thereby focusing less on other tasks and goals, which can have a negative impact on the overall implementation or delivery of the software. As one interviewee put it, “How do you make sure that you don’t end up having a person that’s just fixing technical debt [in order to get the reward].”

Yet another concern that was expressed by several interviewees refers to the possibility of manipulating such reward systems by first introducing a large amount of TD and, after that, refactor it in order to get the reward. “Well, the problem is that it’s very easy to game it. I could introduce a large technical debt item and then refactor it [to get the reward].” Further, an incentive program rewarding the reduction of TD after the TD has first been identified, suppose that in order to get the reward, the reduction of TD must initially be identified and tracked. This suggests that even if the TD is recognized before the manager has identified it, the practitioners could postpone refactoring the TD to get the reward. As one interviewee said, “If you see the debt going up, and you fix it directly, you don’t get any reward. I mean, it’s easy to cheat.”

Taken together, if an incentive program for TD remediation should be introduced, it is important that such a program be carefully designed to avoid counterproductive results that instead generate even more TD; and it is important to design it in an impartial way that is not easy to manipulate.

19.5.3. Forcing mechanisms

The third research question (RQ3) aims to assess the forcing mechanism to keep the level of TD down and how software engineering practitioners perceive this TD management strategy.

19.5.3.1. Survey results.

When assessing whether the respondents are being forced to keep the level of TD down, the result from the second statement (ST3) in Fig. 4 shows that 15.6% of the respondents agree to some extent with this statement and that 84.4% of the respondents disagree to some extent with being forced to keep the level of TD down.

المنارة للاستشارات

www.r

19.5.3.2. Effective or desirable strategies.

None of the interviewed companies had any forcing rules or requirements related to the management of TD. Notably, all the companies had other forcing rules related to their software development process such as, for example, following code standards (by, e.g., code reviews), documentation requirements, and performing tests. These rules applied primarily to specified mandatory activities and requirements that had to be fulfilled for the delivery to be viewed as complete and further activities to take place.

Even if the TD and its negative effects are known to the software engineers, it can be difficult to get the time and budget from the management side for refactoring the software. Here, a positive side to a forcing TD management strategy was described in terms of empowerment since such a strategy would give authority to the practitioners to conduct mandatory TD remediation tasks without having to argue and motivate the action to the managers. For example, one interviewee said, "I think the motivation is very high to take care of all technical debt, but we don't have the funding to do it. It's no problem motivating the software engineers to fix it, they really want to do it, but we don't have the funding most of the time. I would like the forcing part stating that you actually need to fix this." Yet another finding was that the forcing of TD remediation activities seams to become more vital for companies adopting shared ownership of their software product portfolio, where several teams collaborate on the same software without having strict ownership of the components. Even if the different teams had guidelines which, to some extent, addressed the management of TD, there was no uniformity among the teams regarding how they managed TD since the used guidelines were based only on recommendations without any direct obligations or forcing mechanisms. For example, one interviewee said, "Since we have shared ownership of the code, then we need to have a certain set of common rules to follow to have a good look and feel of the code independent of which team that was in there last. And these rules need to be forced by someone."

No direct negative effects of having forcing rules or requirement related TD were identified in the study. However, several concerns about how such TD managing forcing rules should be designed and implemented was raised by the software practitioners. For example, when adopting a forcing strategy, it is vitally important to define which rules and measurements and responsibilities should be applied by whom together with guidelines addressing how these goals can be achieved. For example, one interviewee said, "You need good rules that people understand, and they need to understand what they shall do." This view was echoed by another interviewee who stated, "But without measurements, you don't have anything to go on. And if everybody is aware of they are being measured, then they will not mess up".

19.5.3.3. Tactics for implementing a forcing strategy.

Another view on the enforcement of TD activities was described as a transition from an encouraging strategy to a forcing strategy.

It was recommended by several interviewees that a company first should focus on encouraging initiatives, and if such a strategy were conceived as not enough, this forcing

المنارة للاستشارات
www.m

strategy could be implemented. To directly implement a forcing strategy was not recommended by the interviewed companies. One company described their history going from an encouraging strategy to adopting a forcing strategy (however, this was not forcing of TD management): “It depends on the scale on the organization. A couple of years ago, we didn’t force that much. We were encouraged, and that was because we had some sort of thought of ownership, and I would say product pride in our product, and that was our sort of encouragement. It was your baby, and you wanted to be proud of it, and that’s driving bits.”

19.5.4. Penalizing Disincentives

The fourth research question (RQ4) set out to investigate how common penalizing disincentives are to keep the level of TD down and how software engineering practitioners perceive this TD management strategy.

19.5.4.1. Survey results.

Looking at the third statement (ST2) in Fig. 4, it is apparent that 17.5% of the respondents agree to some extent with being explicitly penalized when not keeping the level of TD down, and thus 82.5% of the respondents state that they are not penalized if they don’t.

19.5.4.2. Effective or desirable strategies.

Even if the survey showed that respondents are being penalized (individually or team-wise), none of the interviewees in the study were familiar with any penalizing activities within their companies, causing, for example, monetary fines and salary reductions. All of the interviewees had direct negative attitudes toward implementing a TD management strategy based on penalizing practitioners or teams who do not succeed in keeping the level of TD down. Commenting on penalization strategies, one of the interviewees said, “No, I don’t like, for example, lowered salary because we don’t meet the expectations.”

19.5.4.3. Tactics for penalizations.

Several different concerns were raised about a penalizing strategy for managing TD. Initially, to be able to penalize an undesired behavior where, for example, TD is introduced (deliberate or un deliberate) or when identified TD are not refactored in a wanted way or within a specified period, there must be clear rules for the software practitioners to follow. These rules can potentially be the same type of rules as the rules formulated in the forcing strategy, but they can also be of a different character.

However, there are several different issues to take into consideration if a penalizing strategy should be implemented to facilitate the management of TD. Several interviewees raised the importance of establishing fair, adequate, and succinct rules that should be clearly conveyed, understood, and followed by all the concerned employees.

المنارة للاستشارات

www.n

Initially, there must be a general understanding and definition of TD among all concerned employees. An interviewee addressing this issue said, “For penalizing, then people are going to sweep it under the carpet and say, “Oh, I don’t know about any technical debt’.”

Further, there also needs to be someone to be accountable for defining the rules and someone who should be responsible for tracking, identifying, and penalizing the individual employee or the team. The justice of the rules must also be addressed appropriately, meaning that there must be clear rules on “whom” to penalize. One example to illustrate this can be when a project manager orders a developer to introduce TD intentionally due to time restriction. The penalizing rules must be clear on whom to penalize in such a situation: the person who orders the introduction of TD or, for example, the developer who actually implements the suboptimal solution. One of the interviewed developers highlighted such a scenario by describing that TD sometimes has to be intentionally introduced because of a requirement coming from outside the one’s team: “The penalizing part, it’s often just like it’s not in your control if you want to keep the debt down fully, and some of this is probably your fault, but often it’s something from outside your team that’s saying: ‘No, you have to do this now [introduce the TD delibertely].’”

19.6. Discussion and limitations

Since no research to date was found in the SE research field on the question of how different TD management strategies can be applied to keep the level of TD down, this research provides novel results.

In total, four main strategies were identified in the initial literature research, which was particularly interesting when portraying how software managers can influence and impact the software engineers’ attitudes and working behaviors with TD.

As illustrated in Fig. 5, we propose a model describing the different identified and investigated TD management strategies. The model spans four quadrants and is named “The Four TD Management Quadrants,” which outlines that TD managing strategies can be either of an incentive/disincentive character having a focus on either a desired or undesired behavior.

This model offers an exploration of different TD management strategies, with its different strategies and tactics, and it also describes both how the strategies relate to each other and how they differ. This model can assist managers when making decisions on which strategy to adopt and also support transition plans, shifting from one strategy to another.

Among the different studied strategies, the result shows that today's software companies most commonly use a TD management strategy based on the encouragement of employees, where 60% of the respondents in the survey state that they are, to some extent, encouraged to keep the level of TD down. Meanwhile, the other investigated strategies, such as using a strategy based on forcing mechanisms or the adoption of incentive or disincentive programs, were rarely used by the companies.

Further, among the investigated companies, there was a strong belief that both the encouraging and the reward TD management strategy would be valuable to decrease

المنارة للاستشارات

www.r

further the amount of TD in the software; meanwhile, the forcing and penalizing strategies were not considered as desired and constructive strategies.

One motivating finding is that practitioners conceive that the attitudes and mindset toward TD remediation tasks from their managers have a significant impact on the way they address TD. Further, and perhaps the most striking results in this study are that a TD managing strategy based on encouragement has a significant impact on the way practitioners work with TD.

Though, since the result shows that still quite a lot of the respondents to some extent do not agree with being encouraged (40%) to focus their effort on TD remediation tasks, this result shows that there is an unfulfilled potential for managers to impact how practitioners can reduce TD by adopting a TD management strategy based on encouragement and without having to introduce forcing mechanisms or strategies based on rewards or penalizations.

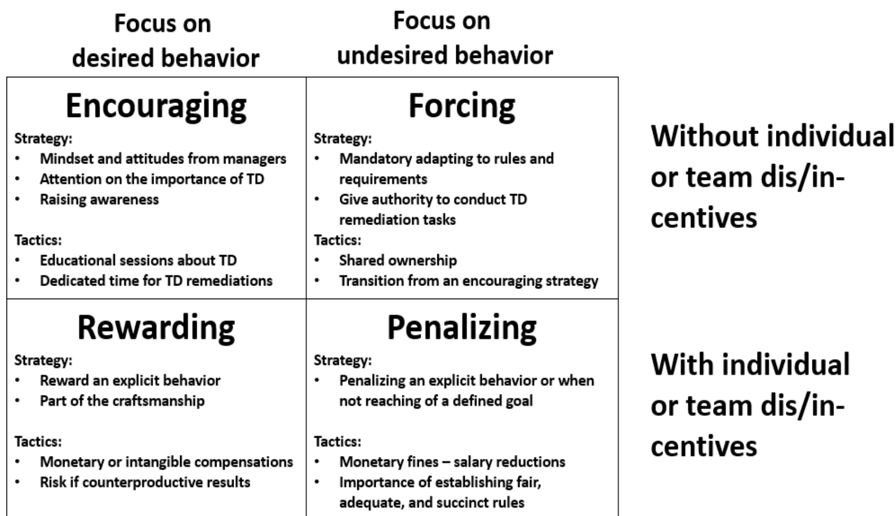


Figure 5: The Four TD Management Quadrants Model

19.7. Implications for future practice

Based on the findings in this study, we summarize the following five recommendations for how different TD management strategies could be operationalized in software developing companies:

Recommendation 1: Having an explicit TD management strategy can impact the amount of TD in the software significantly. However, when proposing and implementing such a strategy, managers should consult with concerned stakeholders to understand which strategy is most effective and appropriate for use.

Recommendation 2: A TD management strategy based on encouraging the software engineers to keep the level of TD down is a good starting point that must not require much effort from a management perspective. Such an encouraging strategy should be based on

المنارة للاستشارات

www.r

a leadership empowering participation in TD remediation activities where the goal of the strategy should be inspiring and having a clarity of purpose, value, and direction.

Initially, the managers can recognize and encourage the desired way of addressing TD in the software, which will reinforce the employee's understanding of what is expected. Other encouraging activities can be to provide education about TD and thereby to raise awareness about its importance and its negative consequences.

Recommendation 3: When introducing a rewarding strategy, it is recommended to design it carefully to ensure justice. It is also recommended to consider incentives of both non-financial and tangible rewards initially and, thereafter, evaluate the outcome of the two different reward strategies to make sure the effect will not be counterproductive to other goals.

Recommendation 4: Specifically, companies adopting shared ownership of their software product portfolio and already having explicit forcing rules or requirements for the development process, it is also recommended to integrate rules that address TD. However, these forcing rules should be well designed together with guidelines and examples demonstrating how the goals can be achieved.

Recommendation 5: It is advised to start implementing a TD management strategy and, thereafter, evaluate the strengths and weaknesses of the chosen strategy. We also recommend companies to use "The Four TD Management Quadrants" model to assist when making decisions related to transitions from one strategy to another.

19.8. Threats to validity

Several important threats to validity necessitate a cautious interpretation of the results of the present study. We have chosen a classification scheme in order to distinguish between different aspects of validity and threats to validity provided by Runesson and Höst [73]. This scheme includes four aspects of validity: construct validity, internal validity, external validity, and reliability.

Construct validity reflects the extent to which the operational measures that are studied represent what the researchers have in mind and what is investigated according to the stated research questions [73]. In a survey, this is commonly one of the main threats to validity, as the respondents might interpret the survey questions and other terms differently. To mitigate this threat, we provided the respondents with the following description of TD before answering the questions: "Technical Debt is a metaphor that describes a real-life phenomenon, and it provides a way of talking and reasoning about difficulties related to software development and software maintenance. Below is a brief description of what Technical Debt is: Technical Debt (TD) is usually described as the non-optimal code or other artifacts related to software development that gives a short-term benefit but causes a long-term extra cost during the software life-cycle." Further, this study could possibly suffer from internal validity by affecting our ability to explain the phenomena that we observed [21] accurately. However, to mitigate this threat, we triangulated the findings from the survey by conducting follow-up interviews validating the derived results. Additionally, to minimize the threat of the respondents

المنارة للاستشارات

www.r

misunderstanding the different strategies in the survey, we initially conducted a pilot study with an industry practitioner, and the understanding of the terms was also addressed during the follow-up interviews. External validity focuses on the extent to which it is possible to generalize the findings. There is always a risk in surveys that the sample is biased, and for this topic, a potential threat refers to the demographic and cultural distribution of response samples. As reported in Section 4.2, we mainly investigated companies from the Scandinavian area, which can have a cultural impact on respondents' experiences and views on penalizing, rewarding, forcing, and encouraging actions from management. The results could, therefore, potentially be different in other cultural or geographical areas. Thus, further work is needed to replicate the results in other geographical areas and other software development cultures. However, to mitigate this validity issue, we attempted to enlarge the respondents' sample by inviting additional participants globally via LinkedIn. Without replicating this study to other countries, it is not possible to confirm that this study is fully generalizable. Reliability addresses whether a study would yield the same results if other researchers replicated it. In this sense, triangulation is important [73], and to mitigate this threat, we have employed source triangulation (several companies and several professional roles), methodological triangulation (quantitative analysis based on survey and qualitative analysis based on interviews), and observer triangulation (all authors participated in the analysis). With regard to the recommendations, some limitations need to be acknowledged. Since only the encouraging strategy was to some extent adopted by the interviewees, the recommendations are based on both the interviewees' experience with strategies focusing on issues other than TD and on their opinions on the different strategies.

19.9. Conclusion

This study set out to investigate empirically how common different management strategies are when managing TD and to investigate how software practitioners perceive such TD management strategies. More specifically, we study how software management influences the way software practitioners work with TD, for example, by continuously encouraging and rewarding those who focus on TD remediation and limitation activities. Yet another TD management strategy we examine in this study is based on penalizations and forcing mechanisms. The results show that software practitioners are not commonly rewarded, penalized, or forced to keep the level of TD down. However, 60% of the respondents state that they are, to some extent, encouraged to keep the level of TD down. Further, the result shows that a TD management strategy based on encouraging activities is described as having a significant impact on software engineers' attitudes and behaviors when addressing TD. Since about 40% of the survey respondents state that they are not directly encouraged from managers to keep the level of TD down, this indicates that there is considerable unfulfilled potential to influence how software practitioners further can limit and reduce TD by adoption of a TD management strategy based on encouraging activities where, for example, the concept of TD is acknowledged and recognized more broadly. Finally, this study also contributes to a TD management quadrant model describing four different TD management strategies and its tactics together with recommendations on how to implement such strategies in practice.

المنارة للاستشارات

www.m

المنارة للاستشارات
www.m

20. Answering the Thesis' Research Questions

This chapter summarizes the main findings with respect to the stated research questions in this thesis (presented in chapter 4).

20.1. RQ1 – Consequences of TD in today's software industry

The first main research question in this thesis sought to *empirically* study and understand *in what way* and *to what extent* TD influences today's software development work, specifically with the intention of providing more *quantitative* insights into the field. More specifically, we explore the negative effects of TD from four different perspectives, using four sub-questions.

Taken together, the findings show that TD can have a significant negative impact on several different areas within the software development field. In today's competitive environment, software development speed is often an important factor, and companies strive both to reduce development time, shorten the time to market, and increase productivity. However, our results show that software suffering from TD has a significant negative impact on the developer productivity and the results also show that TD causes negative consequences on several different quality attributes which left unchecked can stifle a company's ability to innovate and grow by limiting the resources that are available to implement new features or new software. Further, the developers working with software suffering from TD are also negatively affected, and our results indicate that TD has an influence on developer morale. When comparing different types of TD, our results show that ATD was the most frequently encountered type of TD and also that ATD generates the most negative impact on daily software development work. Collectively, the results suggest that software companies need to understand the burning issues of TD and that TD can affect the future of their software and their investments. It is important that companies recognize the potential long term and far-reaching negative effects of TD and deliberately avoid introducing TD in the first place.

20.1.1. RQ1.1 – What is the negative impact of architectural TD?

In the study in Paper A (chapter 9), we found that the negative impact of ATD can lead to severe and costly maintenance overheads that, in the long run, can diminish an organization's ability to innovate and evolve. ATD can also result in restricted flexibility, decreased reliability, and performance degradation.

Paper D (chapter 12) investigates the impact of ATD on daily software development work, and we found that ATD has a significant negative impact on software practitioners' daily work. Our result shows that practitioners experience that ATD has the highest negative impact on daily software development work. Moreover, we provide evidence that does not support the commonly held belief that ATD increases with the age of software. We show that despite different responsibilities and working tasks of software professionals, ATD negatively affects all roles without any significant difference between roles.

المنارة للاستشارات

www.r

Further, the results indicated a linear correlation between the level of the negative effect of complex architectural design and how often the respondents encounter a limited ability to add new features. These findings confirm that it is important to pay extra attention to the architectural design of the software, in order for the software to be able to grow in the future.

20.1.2. RQ1.2 – What is the negative impact on software quality due to TD?

Software companies need to produce high-quality software and support continuous and fast delivery of customer value both in the short and long term. However, this can be hindered by compromised software quality attributes that have an important influence on the overall software development lifecycle.

The result in paper B (chapter 10) shows that practitioners identified a negative impact on software quality due to TD, where maintenance difficulties, a limited ability to add new features, restricted reusability, poor reliability, and performance degradation issues are the quality issues that have the most negative effect on the software development lifecycle. Further, when examining how frequently software practitioners encountered these compromised quality attributes due to experiencing TD, the results show that maintenance difficulties are most frequently encountered by the respondents.

Moreover, there is a general view that quality issues increase over time during the software lifecycle. However, our findings do not support such view, meaning that our result did not show any statistically evidence that the negative effects of TD increase with the age of the software. Contrary, our result showed that all of the investigated quality issues are frequently encountered even for systems with age less than two years. This finding shows that the payment of interest is early introduced and persist during the whole software lifecycle. Taken together, these findings provide strong empirical confirmation that software companies need to manage TD and reverse TD early in the lifecycle or potentially face quality issues in the future.

20.1.3. RQ1.3 – What is the negative impact on development productivity due to TD?

Software systems suffering from TD cause an extensive amount of wasted working time since practitioners are forced to perform additional activities, which would not be necessary if the TD was not present (such as e.g. performing additional testing, source code analysis, and refactoring). In this thesis, we have used this amount of wastage as a proxy for productivity, and our findings show that TD causes an extensive amount of waste of working time for the concerned practitioners and thereby having a negative impact on the productivity. The findings in paper C (chapter 11) show that, on average, 36% of all development time is *estimated* to be wasted due to TD and that all investigated software roles (several different roles) are heavily affected by the interest of TD. Further, when specifically studying productivity using data from *developers reporting* wastage of

المنارة للاستشارات

www.r

time due to TD in paper E (chapter 13), the result shows that, on average, 23% of their time is wasted.

Taken together, the findings suggest strong recommendations for software companies to focus further on continuously undertaking refactoring initiatives of TD to keep the amount of TD at bay on an ongoing basis and thereby also potentially increase the overall developer productivity.

20.1.4. RQ1.4 – What is the negative impact on developers' morale due to TD?

TD has a negative impact on developers' morale. Our results in paper F (chapter 14) indicate that the occurrence of TD reduces developers' morale since the presence of TD hinders the developers' progress and reduces their confidence. Further, our results imply that proper management of TD increases developers' morale since it enables the developers to perform their tasks better and to improve software quality in the future. The results also suggest that proper TD management leads to a virtuous cycle where the right culture and TD prevention mechanisms reinforce each other, leading to less waste of time, followed by a continuous increase in the developers' morale and productivity. To conclude, these findings show that developers consider TD and its management as important factors influencing their working progress and future development activities, and this result encourages software companies to also consider the human and organizational consequences of TD more seriously.

20.2. RQ2 – Initiatives to reduce the negative effects of TD in today's software industry

The second main research question in this thesis set out with the aim of understanding which different initiatives can reduce the negative effects of TD and also which factors are important to consider when implementing such initiatives in the software industry. Taken together, the findings show that there are different initiatives that can be undertaken in order to reduce the negative consequences of TD. First, software practitioners need to understand the negative consequences of TD and, thereafter, identify and further strengthen initiatives with the intension of reducing or avoiding TD. The findings indicate that software companies need to be armed with strategies and proactive management in order to reduce the negative effect of TD. The development of such strategies could include the process of tracking and prioritizing TD using a systematic approach and the usage of supporting guidelines or standards. Further, having a TD management strategy based on "encouragement" can significantly impact the amount of TD in the software, since our findings show that practitioners conceive that the attitudes and mindset toward TD remediation tasks from their managers have a significant impact on the way they address TD.

المنارة للاستشارات

www.r

20.2.1. RQ2.1 – What impact do TD prevention initiatives have on software development?

Since TD has a significant negative impact on software development, our results show that it is important to actively prevent introducing TD in the software in the first place. Our results show for instance, that there is a correlation between the amount of TD and productivity (paper C and paper E), which infers that if the introduction of TD can be prevented in the first place, software productivity may increase. Further, our results in paper E (chapter 13), also show that developers are frequently forced to introduce new TD, due to already implemented TD, meaning that preventing the introduction of TD in the first place will also decrease the later introduction of additional TD.

Further, the result in paper I (chapter 17), show that, e.g., adopting regulations and guidelines such as the ones used in the safety-critical software domain, strengthen the implementation of both source code and architecture and thereby initially limit the introduction of TD. Moreover, since we also in paper F (chapter 14) found a correlation between the amount of TD and the developer morale, this also stresses the importance of keeping the level of TD down early and throughout the whole software lifecycle.

20.2.2. RQ2.2 – What impact do TD remediation initiatives have on software development?

Several of the studies in this thesis highlight the importance of iteratively and continuously conduct TD refactoring activities. Our result shows for instance that there is a correlation between TD and productivity (paper C and paper E), which infers that if TD is remediated, the developing productivity may increase.

The findings in paper F (chapter 14) show that conducting TD remediation and refactoring tasks are considered as motivating activities for developers, as they enable them to increase the quality of software artifacts, which made them feel confident about their products.

However, conducting TD refactoring tasks in order to reduce the negative consequences TD causes, may sound easy and obvious to prioritize and perform, but it can often be quite challenging, and the type of software may also have an impact on the possibility to perform TD remediation activities. For instance, the findings in paper J (chapter 18), indicate that the pressure of delivering customer value and meeting delivery deadlines forces the software teams to down-prioritize TD refactorings continuously, in favor of implementing new features rapidly. Further, we also found that, e.g., safety-critical regulations may constrain the possibility of performing TD refactoring activities (paper I), and when studying the impact of introducing incentive programs in paper K (chapter 19) for, e.g., encouraging or rewarding TD remediation activities, our results show that such a program must be carefully designed to avoid counterproductive results that instead generate even more TD.

المنارة للاستشارات

www.r

20.2.3. RQ2.3 – What impact do TD tracking initiatives have on software development?

When TD is present in the software, the only significantly effective way of reducing it is to refactor it. However, before the refactoring can take place, first the TD items need to be tracked and prioritized.

Our results show that TD tracking initiatives have an impact on software development, where it serves to, e.g., monitor TD and its interest over time. However, according to the results in paper G (chapter 15), on average, only 26% of software practitioners use tools to track TD, and only 7.2% of them created a systematic process for doing so. In addition, the results in paper J (chapter 18) also show that even if the used backlogs in today's industry includes TD items to some extent, to support the prioritization process and assist in cost and benefit analyses, etc., it is common that the identified TD items often are managed in so-called "shadow backlogs." The results further indicate that the TD items not being tracked or being managed in "shadow backlogs," potentially may result in the TD items being overlooked during the prioritization process, with the result that the TD items will remain in the software.

Further, to assist the tracking of the TD process for practitioners, we propose a Strategic Adoption Model (SAMTTD) based both on the evidence collected across our studies and in combination with current literature. The model can be used by practitioners to assess their TD tracking process and to plan the next steps to improve their tracking processes.

20.3. RQ3 – Factors affecting TD management in context-specific domains

The third main research question in this thesis sought to study TD in two different domains, a software startup company domain, and a safety-critical software domain. The motivation for studying TD in a software startup context is based on the fact that these types of companies commonly operate under high uncertainty and that their development process commonly is less strict compared to more mature software development companies and thereby potentially alter or expand on the general view of TD.

On the contrary, companies developing software for the SCS domain have commonly both a strict and mature software development process they must adhere to. Taken together, our results indicate that these different types of companies address and manage TD differently, and this illuminates that management strategies of TD also may take into account the context the software companies operate within.

20.3.1. RQ3.1 – What are the challenges and benefits of deliberately introducing TD for software startups?

There are both challenges and benefits of deliberately introducing TD for software startups. The research presented in paper H (chapter 16), has been able to identify several benefits of deliberately introducing TD in a software startup context, such as, e.g., the ability to cut development time, and thereby enable fast feedback from customers and

المنارة للاستشارات

www.r

increase revenue. Another benefit is the preservation of startup capital since startup companies typically have less money in the early stages. Intentional TD also allows startups to remain flexible. When they do not spend large amounts of money or time developing new features, they are more willing to discard them and alter the product significantly as and when needed. Thus, TD allows them to make more objective decisions. However, despite the benefits of intentional TD, we also identified challenges, since introduced TD would eventually need to be fixed, and not addressing the TD issues can cause product failure or business disruption. Another challenge of deliberately introduce TD is the increased risk, the potential productivity loss, and the reduced scalability it often introduces.

20.3.2. RQ3.2 – What impact can regulatory requirements have on TD management in a safety-critical software context?

Regulatory requirements have an impact on TD management in a safety-critical software (SCS) context, from several different perspectives.

The results in paper I (chapter 17) show, for example, that performing TD refactoring tasks in safety-critical software requires several additional activities and costs, compared to non-safety-critical software, and thereby risk being postponed or ignored. The safety critical regulatory requirements in SCS can also force software companies to perform suboptimal work-around solutions that are counterproductive to achieving a high-quality software since the regulations constrain the possibility of performing optimal TD refactoring activities. Since the refactoring of TD in these types of software systems requires both extra activities and costs, it is of vital importance to implement these solutions as optimally as possible from the very beginning.

المنارة للاستشارات

www.r

21. Future Research

The studies included in this thesis contain suggestions for future work. However, based on the synthesis of the results from this thesis' studies, several different opportunities for future research in new areas and domains are suggested.

As illustrated in Figure 1, two main TD tracks of future research are outlined, where the first track focuses on TD in Machine Learning (ML) applications and the second track focuses on Process Debt.

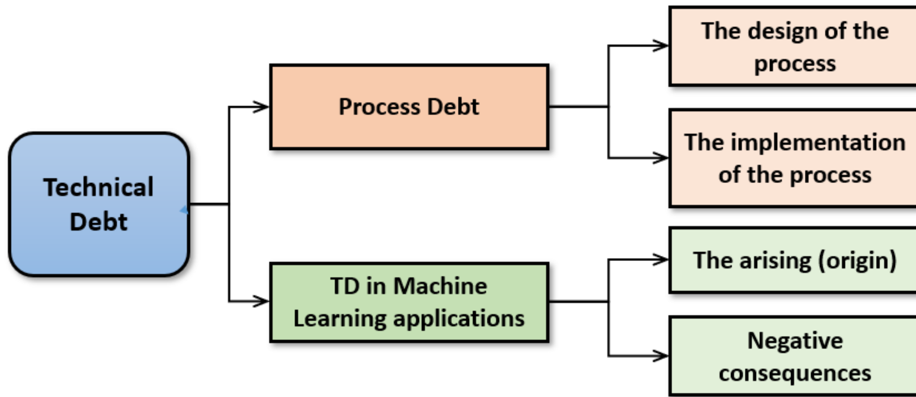


Figure 1: Possible directions for future research.

21.1. Technical Debt in Machine Learning applications

There has been substantial research undertaken on the role of investigating TD in *traditional* software development, but very few studies have empirically studied the TD concept within artificial intelligence (AI) and ML applications.

The development of AI-based software applications based on ML algorithms is today increasing at a rapid rate. In comparison with traditional software systems, all ML algorithms feed on large pools of data, where the data is the prime constituent of the AI application's success. The data sets used in the ML algorithms need to be of exceptionally high quality, as even minor issues can have a negative impact on the output from the very beginning. This means that any AI application using ML algorithms will only be as good as the quality of the data used in the pipeline. However, since the data pipeline is commonly continually changing to accommodate additional data sources, the data ingestion rate is often unpredictable, and the quality of the data frequently requires auditing and cleaning. More profound insights into how TD arise in ML applications, together with its negative consequences in such applications, would be beneficial for the software engineering field.

المنارة للاستشارات
www.m

21.2. Process Debt

The software development process is crucially important when developing software. Very few studies today address TD in relation to the implemented software development process both in terms of the design of the process and also how the process is carried out in practice. Since this process has a significant impact on the software product being developed [269], this type of debt has a potentially significant impact and would, therefore, benefit from being further investigated.

المنارة للاستشارات

www.r

22. Conclusion

Returning to the goal posed at the beginning of this thesis, it is now possible, based on empirical data, to state the negative effects TD has on software development from several different aspects, encompassing technical, financial, and human perspectives.

The studies in this thesis that investigate different types of TD show that ATD is of significantly high importance to software companies and that, among the different types of TD, ATD is the most commonly encountered type of TD. Furthermore, this study shows that ATD has the greatest negative impact on daily software development work, as estimated by all the different software professional roles surveyed.

Most commonly, in the available academic literature, the negative effects due to TD are described in terms of maintenance complications and evolvability issues of the software. However, this investigation has been able to show that TD also has negative effects on other software quality attributes, which are thus important and need to be addressed. In addition to causing maintainability complications and a limited ability to add new features, this investigation also shows that TD has a negative effect on software quality attributes, such as restricted reusability, poor reliability, and performance degradations, by providing a quantified estimation of the negative impact of each of the compromised quality attributes. Furthermore, this thesis broadly supports the common view presented in academic literature in this area, where our study empirically shows that almost all the investigated types of TD cause maintenance difficulties as the most frequently encountered quality issue. When studying the frequency of encountering maintenance complications with respect to the age of the software, this study did not find any evidence for the generally held belief that maintenance complications increase with the age of the software.

This Ph.D. thesis also shows that software suffering from TD caused software practitioners to perform additional time-consuming work activities. The time spent on these activities is referred to as *wasted time* in this thesis, where we also use this variable as a proxy for productivity. By empirically assessing the amount of wasted time, using data based from both single *estimations* and repeated *reportings* by software practitioners, this study shows that software practitioners spend an extensive part of their software development working time on these activities. One striking result emerging from this study is that, on average, software practitioners (from several different roles) *estimated* that 36% of all software development time is wasted because of paying the interest due to TD. Furthermore, when specifically studying the amount of wasted time *developers report* that they waste due to TD, our findings show that they waste, on average, 23%.

Moreover, when studying which different additional working tasks, this wasted time is spent on, the following activities were identified: performing additional testing, conducting additional source code analysis, and performing additional refactoring. This study also found that all different software professional roles are affected by TD. Further, the results also reveal that, on a quarter of the occasions where developers encounter TD, they are forced to introduce additional TD due to the already existing TD. This burden of being forced to introduce additional TD demonstrates the contagiousness of TD, inferring that the interest cost of the TD might potentially grow exponentially.

للاستشارات

www.r

When studying how TD affects the developers' morale, it is evident that working with software experiencing TD can reduce their morale, which can be described in terms of the issue that the TD hinders them from performing their tasks and achieving their goals. On the other hand, the results clearly suggest that proper management of TD increases developers' morale.

This thesis also includes studies of the effects of TD in two different context-specific domains; the startup domain and the safety-critical domain. When studying TD from a startup company perspective, it is evident that software startups differ significantly from mature organizations in how they accumulate and manage TD. The findings show, e.g., that intentionally introducing TD allows startups to cut development time, enable faster feedback and increased revenue, preserve their resources, and decrease risk. Further, while there are several similarities between non-safety critical software and safety-critical software (SCS), our results show that there are also several major differences. One of the differences is that that the heavy SCS regulations can potentially force companies to perform work-around solutions that are counterproductive to achieving a maintainable and evolvable software since the regulations constrain the possibility of performing optimal TD refactoring activities. The overall results of this thesis empirically demonstrate that software encountering TD in general, and ATD specifically, causes several different negative effects, from both the technical, financial, and social perspectives. The findings in this thesis further demonstrate that the consequences of TD can, over time, result in issues such as project delays, software quality complications, high defect rates, reduced developer morale, and very low developer productivity. In the long run and left unchecked, these issues can seriously impede organizations' ability to grow and innovate by impeding innovation and expansion of their software systems.

This thesis also identifies several initiatives that can be undertaken in order to reduce the negative effects of TD. The findings show that software development organizations need to understand and continuously prioritize (e.g., by logging the TD items in the official backlog) and remediate TD in both newer projects and in more mature software. Implementing prevention mechanisms such as, e.g., introducing clear guidelines together with regulations, can also have a positive impact on avoiding TD in the first place. Moreover, one key necessity for reducing the potential negative effects of TD is to track it. However, this study revealed that only 7% of the investigated companies had applied a systematic tracking process for TD. This indicates that software companies need to be armed with strategies and proactive management to enable them to track and manage the interest of TD. Such strategies could result in better, more informed decisions to balance the accumulation and the repayment of TD. Further, the result shows that a TD management strategy based on encouraging activities is described as having a significant impact on software engineers' attitudes and behaviors when addressing TD. Since 40% of the surveyed software engineers state that they are not directly encouraged by managers to keep the level of TD down, this indicates that there is considerable unfulfilled potential to influence how software practitioners can further limit and reduce TD.

المنارة للاستشارات

www.r

المنارة للاستشارات

المنارة للاستشارات
www.m

23. Bibliography

[1] Elsevier. "CRediT author statement, https://www.elsevier.com/authors/journal-authors/policies-and-ethics/credit-author-statement."

[2] C. Page, Software Engineering: Larsen and Keller Education, 2017.

[3] M. Reddy, "Chapter 4 - Design," API Design for C++, M. Reddy, ed., pp. 105-150, Boston: Morgan Kaufmann, 2011.

[4] P. Avgeriou, P. Kruchten, I. Ozkaya, and C. Seaman, “Managing Technical Debt in Software Engineering (Dagstuhl Seminar 16162),” Dagstuhl Reports, vol. 6, no. 4, pp. 110-138, 2016.

[5] A. Martini, J. Bosch, and M. Chaudron, “Architecture technical debt: Understanding causes and a qualitative model,” In: Proceedings - 40th Euromicro Conference Series on Software Engineering and Advanced Applications, SEAA 2014, 2014, pp. 85-92.

[6] C. Giardino, N. Paternoster, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software Development in Startup Companies: The Greenfield Startup Model,” IEEE Transactions on Software Engineering, vol. 42, no. 6, pp. 585-604, 2016.

[7] E. Tom, A. Aurum, and R. Vidgen, “An exploration of technical debt,” Journal of Systems and Software, vol. 86, no. 6, pp. 1498-1516, 2013.

[8] E. Lim, N. Taksande, and C. Seaman, “A Balancing Act: What Software Practitioners Have to Say about Technical Debt,” IEEE Software, vol. 29, no. 6, pp. 22-27, 2012.

[9] V. Lenarduzzi, T. Orava, N. Saarimäki, K. Systa, and D. Taibi, “An Empirical Study on Technical Debt in a Finnish SME,” In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), 2019, pp. 1-6.

[10] J. Yli-Huumo, A. Maglyas, and K. Smolander, “The sources and approaches to management of technical debt: a case study of two product lines in a middle-size finnish software company,” Conf. Product-Focused Software Process Improvment, pp. 93-107, 2014.

[11] Z. Codabux, and B. Williams, “Managing technical debt: an industrial case study,” Proceedings of the 4th International Workshop on Managing Technical Debt, pp. 8-15, 2013.

[12] R. O. Spinola et al., “Investigating technical debt folklore: Shedding some light on technical debt opinion,” Managing Technical Debt (MTD), 2013 4th International Workshop on, pp. 1-7, 2013.

[13] C. Fernández-Sánchez, J. Garbajosa, and A. Yagüe, “A framework to aid in decision making for technical debt management,” In: 7th IEEE Workshop on Managing Technical Debt, 2015, pp. 69-76.

[14] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spinola, “Towards an Ontology of Terms on Technical Debt,” In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on, 2014, pp. 1-7.

[15] Z. Li, P. Avgeriou, and P. Liang, “A systematic mapping study on technical debt and its management,” Journal of Systems and Software, vol. 101, pp. 193-220, 2015.

المنارة للاستشارات

www.r

[16] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, “Measure it? Manage it? Ignore it? software practitioners and technical debt,” In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 2015, pp. 50-60.

[17] J. Holvitie, V. Leppanen, and S. Hyrynsalmi, “Technical Debt and the Effect of Agile Software Development Practices on It - An Industry Practitioner Survey,” In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on, 2014, pp. 35-42.

[18] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical Debt: From Metaphor to Theory and Practice,” Software, IEEE, vol. 29, no. 6, pp. 18-21, 2012.

[19] S. McConnell. "Technical Debt. 10x Software Development [cited 2010 June 14]," http://www.construx.com/10x_Software_Development/Technical_Debt/.

[20] M. Fowler. "Technical Debt Quadrant. Bliki [Blog]," Dec, 2015; http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html.

[21] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, “What is social debt in software engineering?,” 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings, pp. 93-96, 2013.

[22] Z. Li, P. Liang, and P. Avgeriou, "Architectural Debt Management in Value-Oriented Architecting," Economics-Driven Software Architecture, pp. 183-204, 2014.

[23] A. Martini, and J. Bosch, “The Danger of Architectural Technical Debt: Contagious Debt and Vicious Circles,” In: Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, 2015, pp. 1-10.

[24] C. Fernández-Sánchez, J. Díaz, J. Pérez, and J. Garbajosa, “Guiding flexibility investment in agile architecting,” In: Proceedings of the Annual Hawaii International Conference on System Sciences, 2014, pp. 4807-4816.

[25] B. Curtis, J. Sappidi, and A. Szynkarski, “Estimating the size, cost, and types of technical debt,” In: 2012 3rd International Workshop on Managing Technical Debt, MTD 2012 - Proceedings, 2012, pp. 49-53.

[26] Z. Li, P. Liang, P. Avgeriou, N. Guelfi, and A. Ampatzoglou, “An empirical investigation of modularity metrics for indicating architectural technical debt,” In: Proceedings of the 10th international ACM Sigsoft conference on Quality of software architectures, Marcq-en-Bareul, France, 2014, pp. 119-128.

[27] I. Ozkaya, R. L. Nord, H. Koziolek, and P. Avgeriou, “Second International Workshop on Software Architecture and Metrics (SAM 2015),” In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, 2015, pp. 999-1000.

[28] P. Kruchten, “Strategic management of technical debt: Tutorial synopsis,” In: Proceedings - International Conference on Quality Software, 2012, pp. 282-284.

[29] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, "A financial approach for managing interest in technical debt," Lecture Notes in Business Information Processing, 2016, pp. 117-133.

[30] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, and P. Avgeriou, “The financial aspect of managing technical debt: A systematic literature review,” Information and Software Technology, vol. 64, pp. 52, 2015.

[31] F. Mishkin, and E. Stanley, Financial Markets and Institutions, seventh ed.: Pearson Prentice Hall, 2012.

المنارة للاستشارات

www.r

[32] N. Brown et al., “Managing technical debt in software-reliant systems,” In: Proceedings of the FSE/SDP workshop on Future of software engineering research, Santa Fe, New Mexico, USA, 2010, pp. 47-52.
[33] B. Curtis, J. Sappidi, and A. Szynkarski, “Estimating the principal of an application's technical debt,” IEEE Software, vol. 29, no. 6, pp. 34-42, 2012.
[34] J. de Groot, A. Nugroho, T. Back, and J. Visser, “What is the value of your software?,” In: Managing Technical Debt (MTD), 2012 Third International Workshop on, 2012, pp. 37-44.
[35] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, “Investigating the impact of design debt on software quality,” In: Proceedings of the 2nd Workshop on Managing Technical Debt, Waikiki, Honolulu, HI, USA, 2011, pp. 17-23.
[36] ISO/IEC, “Systems and software engineering — Systems and software Quality Requirements and Evaluation (SQuaRE) system and software quality models, ISO/IEC FDIS 25010:2011,” 2011, pp. 1-34.
[37] D. L. Parnas, “Software aging,” In: Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on, 1994, pp. 279-287.
[38] L. de Silva, and D. Balasubramaniam, “Controlling software architecture erosion: A survey,” Journal of Systems and Software, vol. 85, pp. 132-151, 2012.
[39] T. Mens et al., “Challenges in software evolution,” In: Eighth International Workshop on Principles of Software Evolution (IWPSE'05), 2005, pp. 13-22.
[40] M. Lindgren, R. Land, C. Norström, and A. Wall, “Key Aspects of Software Release Planning in Industry,” In: 19th Australian Conference on Software Engineering (aswec 2008), 2008, pp. 320-329.
[41] I. Macia et al., “Are automatically-detected code anomalies relevant to architectural modularity?: an exploratory analysis of evolving systems,” In: Proceedings of the 11th annual international conference on Aspect-oriented Software Development, Potsdam, Germany, 2012, pp. 167-178.
[42] N. S. R. Alves et al., “Identification and Management of Technical Debt: A Systematic Mapping Study,” Information and Software Technology, 2015.
[43] C. Sadowski, and T. Zimmermann, “Rethinking Productivity in Software Engineering,” Apress, 2019.
[44] W. Scacchi, “Understanding Software Productivity,,” International Journal of Software Engineering and Knowledge Engineering, vol. 1, no. 3, pp. 293-321, 1991.
[45] E. Oliveira, D. Viana, M. Cristo, and T. Conte, “How have Software Engineering Researchers been Measuring Software Productivity? A Systematic Mapping Study,” Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS), no. 2, pp. 76-87, 2017.
[46] K. D. Maxwell, “Software Development Productivity,” Advances in Computers, vol. 58, pp. 1-46, 2003.
[47] B. Hardy, "Morale: definitions, dimensions and measurement," PhD diss. University of Cambridge, 2010.
[48] N. Zazworka, C. Seaman, and F. Shull, “Prioritizing design debt investment opportunities,” In: Proceedings of the 2nd Workshop on Managing Technical Debt, Waikiki, Honolulu, HI, USA, 2011, pp. 39-42.
[49] L. F. Ribeiro, N. S. R. Alves, M. G. d. M. Neto, and R. O. Spínola, “A Strategy Based on Multiple Decision Criteria to Support Technical Debt Management,” In: 2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2017, pp. 334-341.

المنارة للاستشارات
www.r

[50] C. Seaman et al., “Using technical debt data in decision making: Potential decision approaches,” In: Managing Technical Debt (MTD), 2012 Third International Workshop on, 2012, pp. 45-48.
[51] F. A. Fontana, V. Ferme, M. Zanoni, and R. Roveda, “Towards a prioritization of code debt: A code smell Intensity Index,” In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), 2015, pp. 16-24.
[52] Y. Guo, R. Spínola, and C. Seaman, “Exploring the costs of technical debt management – a case study,” Empirical Software Engineering, pp. 1-24, 2014.
[53] J. Carriere, R. Kazman, and I. Ozkaya, “A cost-benefit framework for making architectural decisions in a business context,” In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, pp. 149-157.
[54] A. Martini, and J. Bosch, “An empirically developed method to aid decisions on architectural technical debt refactoring: AnaConDebt,” In: Proceedings of the 38th International Conference on Software Engineering Companion, Austin, Texas, 2016, pp. 31-40.
[55] A. Martini, E. Sikander, and N. Medlani, “Estimating and Quantifying the Benefits of Refactoring to Improve a Component Modularity: A Case Study,” In: 2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 92-99.
[56] V. Van, Software Engineering: Principles and Practice: John Wiley & Sons, 2008.
[57] A. A. Chughtai, “Linking affective commitment to supervisor to work outcomes,” Journal of Managerial Psychology, vol. Vol. 28 No. 6, pp. 606-62, 2013.
[58] H. Ghanbari, “Seeking technical debt in critical software development projects: An exploratory field study,” In: Proceedings of the Annual Hawaii International Conference on System Sciences, 2016, pp. 5407-5416.
[59] Y. Luo, A. K. Saberi, and M. v. den Brand, "Safety-Driven Development and ISO 26262," Automotive Systems and Software Engineering: State of the Art and Future Trends, Y. Dajsuren and M. van den Brand, eds., pp. 225-254, Cham: Springer International Publishing, 2019.
[60] N. Paternoster, C. Giardino, M. Unterkalmsteiner, T. Gorschek, and P. Abrahamsson, “Software development in startup companies: A systematic mapping study,” Information and Software Technology, vol. 56, no. 10, pp. 1200-1218, 2014.
[61] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering Research," Guide to Advanced Empirical Software Engineering, F. Shull, J. Singer and D. I. K. Sjøberg, eds., pp. 285-311, London: Springer London, 2008.
[62] C. Pope, and N. Mays, “Qualitative Research: Reaching the parts other methods cannot reach: an introduction to qualitative methods in health and health services research,” British Medical Journal 311, pp. 42-45, 1995.
[63] K. Henwood, "Qualitative Research," Encyclopedia of Critical Psychology, T. Teo, ed., pp. 1611-1614, New York, NY: Springer New York, 2014.
[64] D. M. Mertens, and S. Hesse-Biber, “Triangulation and Mixed Methods Research: Provocative Positions,” Journal of Mixed Methods Research, vol. 6, no. 2, pp. 75-79, 2012.
[65] J. W. Creswell, Research design: qualitative, quantitative, and mixed methods approaches, Fourth, international student ed., Los Angeles, Calif: SAGE, 2014.

المنارة للاستشارات

www.r

[66] R. K. Yin, Case study research: design and methods, London: SAGE, 2014.
[67] R. E. Ployhart, and R. J. Vandenberg, “Longitudinal Research: The Theory, Design, and Analysis of Change,” Journal of Management, vol. 36, no. 1, pp. 94-120, 2009.
[68] F. J. Shull, J. C. Carver, S. Vegas, and N. Juristo, “The role of replications in Empirical Software Engineering,” Empirical Software Engineering, vol. 13, no. 2, pp. 211-218, 2008.
[69] L. M. Pickard, B. A. Kitchenham, and P. W. Jones, “Combining empirical results in software engineering,” Information and Software Technology, vol. 40, no. 14, pp. 811-821, 1998.
[70] M. Shepperd, N. Ajienka, and S. Counsell, “The role and value of replication in empirical software engineering results,” Information and Software Technology, vol. 99, pp. 120-132, 2018.
[71] V. R. Basili, F. Shull, and F. Lanubile, “Building knowledge through families of experiments,” IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 456-473, 1999.
[72] S. Vegas et al., “Analysis of the influence of communication between researchers on experiment replication,” In: Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, Rio de Janeiro, Brazil, 2006, pp. 28-37.
[73] P. Runeson, and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” Empirical Software Engineering, vol. 14, pp. 131-164, 2009.
[74] P. Birmingham, and D. Wilkinson, Using Research Instruments : A Guide for Researchers, Abingdon, Oxon, UNITED STATES: Taylor and Francis, 2003.
[75] K.-J. Stol, and B. Fitzgerald, “A holistic overview of software engineering research strategies,” In: Proceedings of the Third International Workshop on Conducting Empirical Studies in Industry, Florence, Italy, 2015, pp. 47-54.
[76] M. Van Selm, and N. W. Jankowski, “Conducting Online Surveys,” Quality and Quantity, vol. 40, no. 3, pp. 435-456, 2006.
[77] R. Czaja, and J. Blair, Designing surveys: a guide to decisions and procedures, Thousand Oaks, Calif: Pine Forge Press, 2005.
[78] B. A. Kitchenham, and S. L. Pfleeger, "Personal Opinion Surveys," pp. 63-92, London: Springer London, 2008.
[79] B. Barn, S. Barat, and T. Clark, “Conducting Systematic Literature Reviews and Systematic Mapping Studies,” In: Proceedings of the 10th Innovations in Software Engineering Conference, Jaipur, India, 2017, pp. 212-213.
[80] B. Kitchenham, “Procedures for performing systematic reviews,” Keele, UK, Keele University, vol. 33, no. 2004, pp. 1-26, 2004.
[81] C. Fisher, Researching and Writing a Dissertation: a guidebook for business students: Prentice Hall, 2007.
[82] G. B. Schaalje, J. B. McBride, and G. W. Fellingham, “Adequacy of Approximations to Distributions of Test Statistics in Complex Mixed Linear Models,” Journal of Agricultural, Biological, and Environmental Statistics, vol. 7, no. 4, pp. 512-524, 2002.
[83] S. Holm, “A Simple Sequentially Rejective Multiple Test Procedure,” Scandinavian Journal of Statistics, vol. 6, pp. 65-70, 1979.
[84] V. Braun, and V. Clarke, “Using thematic analysis in psychology, Qualitative research in psychology, 3(2),” 2006, pp. 77-101.

المنارة للاستشارات

www.r

[85] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, “Coding In-depth Semistructured Interviews Problems of Unitization and Intercoder Reliability and Agreement,” Sociological Methods & Research, 2013.
[86] C. Wohlin et al., Experimentation in software engineering: an introduction: Kluwer Academic Publishers, 2000.
[87] C. Wohlin, “Guidelines for snowballing in systematic literature studies and a replication in software engineering,” In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, London, England, United Kingdom, 2014, pp. 1-10.
[88] J. M. Morse, M. Barrett, M. Mayan, K. Olson, and J. Spiers, “Verification Strategies for Establishing Reliability and Validity in Qualitative Research,” International Journal of Qualitative Methods, vol. 1, no. 2, pp. 13-22, 2002.
[89] J. Miller, “Triangulation as a basis for knowledge discovery in software engineering,” Empirical Software Engineering, vol. 13, no. 2, pp. 223-228, 2008.
[90] P. Brereton, B. A. Kitchenham, D. Budgen, M. Turner, and M. Khalil, “Lessons from applying the systematic literature review process within the software engineering domain,” Journal of Systems and Software, vol. 80, no. 4, pp. 571-583, 2007.
[91] W. Cunningham, “The WyCash portfolio management system, in: 7th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’92),” 1992, pp. 29–30.
[92] H. Van Vliet, Software Engineering: Principles and Practice: John Wiley & Sons, 2008.
[93] A. Martini, J. Bosch, and M. Chaudron, “Architecture Technical Debt: Understanding Causes and a Qualitative Model,” In: Software Engineering and Advanced Applications (SEAA), 2014 40th EUROMICRO Conference on, 2014, pp. 85-92.
[94] T. Besker, A. Martini, and J. Bosch, “Impact of Architectural Technical Debt on Daily Software Development Work - A Survey of Software Practitioners ” In: 43th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Vienna, 2017, pp. 278-287.
[95] T. Besker, A. Martini, and J. Bosch, “Time to Pay Up - Technical Debt from a Software Quality Perspective,” In: proceedings of the 20th Ibero American Conference on Software Engineering (CibSE) @ ICSE17, Buenos Aires, Argentina, 2017, pp. pp. in print. .
[96] Z. Li, P. Liang, and P. Avgeriou, "Chapter 5 - Architecture viewpoints for documenting architectural technical debt," Software Quality Assurance, I. M. S. A. G. Tekinerdogan, ed., pp. 85-132, Boston: Morgan Kaufmann, 2016.
[97] R. L. Nord, I. Ozkaya, P. Kruchten, and M. Gonzalez-Rojas, “In search of a metric for managing architectural technical debt,” In: Proceedings of the 2012 Joint Working Conference on Software Architecture and 6th European Conference on Software Architecture, WICSA/ECSA 2012, 2012, pp. 91-100.
[98] A. Martini, T. Besker, and J. Bosch, “The Introduction of Technical Debt Tracking in Large Companies,” In: 2016 23rd Asia-Pacific Software Engineering Conference (APSEC), 2016, pp. 161-168.
[99] O. Zimmermann, “Designed and delivered today, eroded tomorrow?: towards an open and lean architecting framework balancing agility and sustainability,” In: Proccedings of the 10th European Conference on Software Architecture Workshops, Copenhagen, Denmark, 2016, pp. 1-1.

المنارة للاستشارات

www.m

[100] B. Kitchenham et al., "Systematic literature reviews in software engineering – A systematic literature review," Information and Software Technology, vol. 51, no. 1, pp. 7-15, 2009.

[101] T. Besker, A. Martini, and J. Bosch, "A Systematic Literature Review and a Unified Model of ATD," In: 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2016, pp. 189-197.

[102] B. Curtis, J. Sappidi, and A. Szynkarski, "Estimating the Principal of an Application's Technical Debt," Software, IEEE, vol. 29, no. 6, pp. 34-42, 2012.

[103] Z. Li, P. Liang, and P. Avgeriou, "Architectural Technical Debt Identification Based on Architecture Decisions and Change Scenarios," In: Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, 2015, pp. 65-74.

[104] D. Falessi, M. A. Shaw, F. Shull, K. Mullen, and M. S. Keymind, "Practical considerations, challenges, and requirements of tool-support for managing technical debt," In: Managing Technical Debt (MTD), 2013 4th International Workshop, 2013, pp. 16-19.

[105] C. Wohlin et al., "On the reliability of mapping studies in software engineering," Journal of Systems and Software, vol. 86, no. 10, pp. 2594-2610, 2013.

[106] S. Jalali, and C. Wohlin, "Systematic literature studies: database searches vs. backward snowballing," In: Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement, Lund, Sweden, 2012, pp. 29-38.

[107] J. Webster, and R. T. Watson, "Analyzing the past to prepare for the future: writing a literature review," MIS Quarterly,, vol. 26: 2, 2002.

[108] B. A. Kitchenham et al., "Preliminary guidelines for empirical research in software engineering," Software Engineering, IEEE Transactions on, vol. 28, no. 8, pp. 721-734, 2002.

[109] R. Mo, J. Garcia, C. Yuanfang, and N. Medvidovic, "Mapping architectural decay instances to dependency models," In: Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 39-46.

[110] A. Martini, J. Bosch, and M. Chaudron, "Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study," Information and Software Technology, vol. 67, pp. 237-253, 2015.

[111] P. Avgeriou, P. Kruchten, R. L. Nord, I. Ozkaya, and C. Seaman, "Reducing friction in software development," IEEE Software, vol. 33, no. 1, pp. 66-72, 2016.

[112] E. Poort, "Just Enough Anticipation: Architect Your Time Dimension," IEEE Software, vol. 33, no. 6, pp. 11-15, 2016.

[113] A. MacCormack, and D. J. Sturtevant, "Technical debt and system architecture: The impact of coupling on defect-related activity," Journal of Systems and Software, vol. 120, pp. 170-182, 2016.

[114] N. Ramasubbu, and C. F. Kemerer, "Technical debt and the reliability of enterprise software systems: A competing risks analysis," Management Science, vol. 62, no. 5, pp. 1487-1510, 2016.

[115] S. Bellomo, N. Ernst, R. Nord, and R. Kazman, "Toward design decisions to enable deployability: Empirical study of three projects reaching for the continuous delivery holy grail," In: Proceedings - 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, 2014, pp. 702-707.

المنارة للاستشارات

www.r

[116] R. Kazman et al., "A Case Study in Locating the Architectural Roots of Technical Debt," In: Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on, 2015, pp. 179-188.
[117] L. Xiao, Y. Cai, R. Kazman, R. Mo, and Q. Feng, "Identifying and quantifying architectural debt," In: Proceedings - International Conference on Software Engineering, 2016, pp. 488-498.
[118] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," In: IEEE International Conference on Software Maintenance, ICSM, 2013, pp. 392-395.
[119] K. Schmid, "A formal approach to technical debt decision making," Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures, ACM, 2013, pp. 153-162.
[120] C. Izurieta, G. Rojas, and I. Griffith, "Preemptive Management of Model Driven Technical Debt for Improving Software Quality," In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, Montreal, QC, Canada, 2015, pp. 31-36.
[121] A. Martini, and J. Bosch, "Towards Prioritizing Architecture Technical Debt: Information Needs of Architects and Product Owners," In: Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on, 2015, pp. 422-429.
[122] A. Martini, E. Sikander, N. Medlani, and I. C. Soc, "Estimating and Quantifying the Benefits of Refactoring to Improve a Component Modularity: a Case Study," 2016 42nd Euromicro Conference on Software Engineering and Advanced Applications (Seaa), pp. 92-99, 2016.
[123] D. A. Tamburri, and E. D. Nitto, "When Software Architecture Leads to Social Debt," In: Proceedings - 12th Working IEEE/IFIP Conference on Software Architecture, WICSA 2015, 2015, pp. 61-64.
[124] B. Vogel-Heuser, and S. Rösch, "Applicability of Technical Debt as a Concept to Understand Obstacles for Evolution of Automated Production Systems," In: Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on, 2015, pp. 127-132.
[125] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," In: Proceedings of the 2nd Workshop on Managing Technical Debt, Waikiki, Honolulu, HI, USA, 2011, pp. 35-38.
[126] J. Brondum, and L. Zhu, "Visualising architectural dependencies," In: Proceedings of the Third International Workshop on Managing Technical Debt, Zurich, Switzerland, 2012, pp. 7-14 *.
[127] U. Eliasson, A. Martini, R. Kaufmann, and S. Odeh, "Identifying and visualizing Architectural Debt and its efficiency interest in the automotive domain: A case study," In: Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on, 2015, pp. 33-40.
[128] F. A. Fontana, R. Roveda, and M. Zanoni, "Technical Debt Indexes Provided by Tools: A Preliminary Discussion," In: Proceedings - 2016 IEEE 8th International Workshop on Managing Technical Debt, MTD 2016, 2016, pp. 28-31.
[129] C. Izurieta, I. Ozkaya, C. Seaman, and W. Snipes, "Technical Debt: A Research Roadmap Report on the Eighth Workshop on Managing Technical Debt (MTD 2016)," SIGSOFT Softw. Eng. Notes, vol. 42, no. 1, pp. 28-31, 2017.

المنارة للاستشارات

www.r

[130] D. Reimanis, C. Izurieta, and Ieee, "Towards Assessing the Technical Debt of Undesired Software Behaviors in Design Patterns," 2016 Ieee 8th International Workshop on Managing Technical Debt, International Workshop on Managing Technical Debt, pp. 24-27, New York: Ieee, 2016.

[131] B. Boehm, "Architecture-Based Quality Attribute Synergies and Conflicts," In: Proceedings - 2nd International Workshop on Software Architecture and Metrics, SAM 2015, 2015, pp. 29-34.

[132] F. A. Fontana, V. Ferme, and M. Zanoni, "Towards Assessing Software Architecture Quality by Exploiting Code Smell Relations," In: Proceedings - 2nd International Workshop on Software Architecture and Metrics, SAM 2015, 2015, pp. 1-7.

[133] A. Choudhary, and P. Singh, "Minimizing refactoring effort through prioritization of classes based on historical, architectural and code smell information," In: CEUR Workshop Proceedings, 2016, pp. 76-79.

[134] N. A. Ernst, "On the role of requirements in understanding and managing technical debt," In: Managing Technical Debt (MTD), 2012 Third International Workshop on, 2012, pp. 61-64.

[135] A. Martini, L. Pareto, and J. Bosch, "Towards Introducing Agile Architecting in Large Companies: The CAFFEA Framework," Agile Processes, in Software Engineering, and Extreme Programming, Lecture Notes in Business Information Processing C. Lassenius, T. Dingsøyr and M. Paasivaara, eds., pp. 218-223: Springer International Publishing, 2015.

[136] D. Budgen, B. Kitchenham, and P. Brereton, "The Case for Knowledge Translation," In: 2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, 2013, pp. 263-266.

[137] R. Nickerson, J. Muntermann, U. Varshney, and H. Isaac, "Taxonomy development in information systems: developing a taxonomy of mobile applications, in: 17th European Conference in Information Systems (ECIS '09), Italy," 2009, pp. 1138–1149.

[138] C. Fernández-Sánchez, J. Garbajosa, A. Yagüe, and J. Perez, "Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study," Journal of Systems and Software, vol. 124, pp. 22-38, 2017.

[139] C. Fernández-Sánchez, J. Garbajosa, C. Vidal, and A. Yagüe, "An Analysis of Techniques and Methods for Technical Debt Management: A Reflection from the Architecture Perspective," In: Proceedings - 2nd International Workshop on Software Architecture and Metrics, SAM 2015, 2015, pp. 22-28.

[140] ISO/IEC 25010:201. ""System and software quality models."," 2017-01-22; https://www.iso.org/obp/ui/#iso:std:iso-iec:25010:ed-1:v1:en.

[141] M. F. Aniche, G. A. Oliva, and M. A. Gerosa, "Are the Methods in Your Data Access Objects (DAOs) in the Right Place? A Preliminary Study," In: Managing Technical Debt (MTD), 2014 Sixth International Workshop on, 2014, pp. 47-50.

[142] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," In: Proceedings of the 2nd Workshop on Managing Technical Debt, Waikiki, Honolulu, HI, USA, 2011, pp. 1-8.

[143] D. A. Tamburri, P. Kruchten, P. Lago, and H. Van Vliet, "What is social debt in software engineering?," In: 2013 6th International Workshop on Cooperative and Human Aspects of Software Engineering, CHASE 2013 - Proceedings, 2013, pp. 93-96.

المنارة للاستشارات

www.r

[144] T. Besker, A. Martini, and J. Bosch, “The pricey Bill of Technical Debt - When and by whom will it be paid? ,” In: IEEE International Conference on Software Maintenance and Evolution (ICSME), Shanghai, China, 2017, pp. 13-23.
[145] P. Runeson, and M. Höst, “Guidelines for conducting and reporting case study research in software engineering,” Empirical Software Engineering, vol. 14, no. 2, pp. 131-164, 2008.
[146] R. A. Krueger, and M. A. Casey, Focus groups: a practical guide for applied research, Thousand Oaks, Calif: Sage Publications, 2009.
[147] C. F. Auerbach, L. B. Silverstein, and Ebrary, Qualitative data: an introduction to coding and analysis, New York: New York University Press, 2003.
[148] L. M. Lix, J. C. Keselman, and H. J. Keselman, “Consequences of Assumption Violations Revisited: A Quantitative Review of Alternatives to the One-Way Analysis of Variance "F" Test,” Review of Educational Research, vol. 66, no. 4, pp. 579-619, 1996.
[149] Z. Codabux, and B. Williams, “Managing technical debt: an industrial case study,” In: Proceedings of the 4th International Workshop on Managing Technical Debt, San Francisco, California, 2013, pp. 8-15.
[150] T. Besker, A. Martini, and J. Bosch, “A systematic literature review and a unified model of ATD,” In: Euromicro Conference series on Software Engineering and Advanced Applications (SEAA), 2016.
[151] B. Curtis, J. Sappidi, and A. Szynkarski, “Estimating the size, cost, and types of technical debt,” In: Proceedings of the Third International Workshop on Managing Technical Debt, Zurich, Switzerland, 2012, pp. 49-53.
[152] A. Szykarski. "Ted Theodorpoulos on Managing Technical Debt Successfully, Available from: http://www.ontechnicaldebt.com/blog/ted-theodoropoulos-on-managing-technical-debt-successfully/."
[153] T. Besker, A. Martini, and J. Bosch, “Managing architectural technical debt: A unified model and systematic literature review,” Journal of Systems and Software, vol. 135, no. Supplement C, pp. 1-16, 2018.
[154] N. Mellegård, “Using weekly open defect reports as an indicator for software process efficiency: theoretical framework and a longitudinal automotive industrial case study,” In: Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement, Gothenburg, Sweden, 2017, pp. 170-175.
[155] V. S. Fonseca, M. P. Barcellos, and R. de Almeida Falbo, “An ontology-based approach for integrating tools supporting the software measurement process,” Science of Computer Programming, vol. 135, pp. 20-44, 2017.
[156] M. Murray, and N. Kujundzic, Critical Reflection : A Textbook for Critical Thinking, Montreal, CANADA: MQUP, 2005.
[157] T. Besker, A. Martini, and J. Bosch, “Technical Debt Cripples Software Developer Productivity - A longitudinal study on developers’ daily software development work,” First International Conference on Technical Debt @ ICSE18, 2018.
[158] V. Basili, G. Caldiera, and D. Rombach, “The Goal Question Metric Approach,” Encyclopedia of Software Eng., J.J. Marciniak, ed., pp. 528-532, 19924.
[159] E. Oliveira, D. Viana, M. Cristo, T. Conte, and “How have Software Engineering Researchers been Measuring Software Productivity? A Systematic Mapping Study ” In: Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS), 2017, pp. 76-87.

المنارة للاستشارات

www.r

[160] K. D. Maxwell, “Collecting data for comparability: benchmarking software development productivity,” IEEE Software, vol. 18, no. 5, pp. 22-25, 2001.
[161] R. W. Jensen, Improving Software Development Productivity: Effective Leadership and Quantitative Methods in Software Management: Prentice Hall Press, 2014.
[162] T. Sedano, P. Ralph, and C. e. Péraire, “Software development waste,” In: Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, 2017, pp. 130-140.
[163] A. Martini, and J. Bosch, “On the interest of architectural technical debt: Uncovering the contagious debt phenomenon,” Journal of Software: Evolution and Process, 2017.
[164] A. Martini, T. Besker, and J. Bosch, “Technical debt tracking: Current state of practice a survey and multiple case study in 15 large organizations,” Science of Computer Programming, 2018.
[165] R. J. Eisenberg, “A threshold based approach to technical debt,” SIGSOFT Softw. Eng. Notes, vol. 37, no. 2, pp. 1-6, 2012.
[166] N. Juristo, A. M. Moreno, SpringerLink, and A. SpringerLink, Basics of Software Engineering Experimentation, 1 ed., Boston, MA: Springer US, 2001.
[167] D. F. Morrison, “The optimal spacing of repeated measurements,” Biometrics, vol. 26:281-90, 1970.
[168] R. Core-Team, “R: A language and environment for statistical computing. R Foundation for Statistical Computing,” URL https://www.R-project.org/, 2016.
[169] T. Therneau, and B. Atkinson, “ rpart: Recursive Partitioning and Regression Trees. R package version 4.1-13.

URL https://CRAN.R-project.org/package=rpart,” 2018.
[170] C. B. Seaman, “Qualitative methods in empirical studies of software engineering,” IEEE Transactions on Software Engineering, vol. 25, no. 4, pp. 557-572, 1999.
[171] J. C. Carver, “Towards Reporting Guidelines for Experimental Replications: A Proposal,” In: 1st International Workshop on Replication in Empirical Software Engineering Research (RESER), Cape Town, South Africa, 2010.
[172] D. Graziotin, X. Wang, and P. Abrahamsson, “Do feelings matter? On the correlation of affects and the self-assessed productivity in software engineering,” J. Softw. Evol. Process, vol. 27, no. 7, pp. 467-487, 2015.
[173] A. Ampatzoglou, A. Ampatzoglou, P. Avgeriou, and A. Chatzigeorgiou, “A financial approach for managing interest in technical debt,” Lecture Notes in Business Information Processing, vol. 257, pp. 117-133, 2016.
[174] C. Peterson, N. Park, and P. J. Sweeney, “Group well‐being: morale from a positive psychology perspective,” Appl.Psychol., vol. 57, no. s1, pp. 19-36.
[175] L. McLeod, and B. Doolin, “Information systems development as situated socio-technical change: a process approach,” European Journal of Information Systems, vol. 21, no. 2, pp. 176-191, 2012.
[176] R. Feldt, L. Angelis, R. Torkar, and M. Samuelsson, “Links between the personalities, views and attitudes of software engineers,” Information and Software Technology, vol. 52, no. 6, pp. 611-624, 2010.
[177] F. Fagerholm, and J. Münch, “Developer experience: concept and definition,” 2012 International Conference on Software and System Process (ICSSP), pp. 73-77.

المنارة للاستشارات

www.r

[178] Graziotin, X. Wang, and P. Abrahamsson, “Happy software developers solve problems better: psychological measurements in empirical software engineering,” PeerJ. 2: e289, 2014.
[179] J. Abbott, “Does employee satisfaction matter? A study to determine whether low employee morale affects customer satisfaction and profits in the business-to-business sector,” Journal of Communication Management 7(4):, pp. 333-339, 2003.
[180] C. J. Stowe, “Incorporating morale into a classical agency model: implications for incentives, effort, and organization,” Economics of governance. 10(2), 2009.
[181] D. Damian, and J. Chisan, “An empirical study of the complex relationships between requirements engineering processes and other processes that lead to payoffs in productivity, quality, and risk management,” IEEE Transactions on Software Engineering, vol. 32, no. 7, pp. 433-453, 2006.
[182] R. E. Fairley, and M. J. Willshire, “Iterative rework: The good, the bad, and the ugly,” Computer, vol. 38, no. 9, pp. 34-41, 2005.
[183] L. R. Foulds, and M. West, “The productivity of large business information system development,” Int. J. Bus. Inf. Syst., vol. 2, no. 2, pp. 162-181, 2007.
[184] T. Hall, D. Jagielska, and N. Baddoo, “Motivating developer performance to improve project outcomes in a high maturity organization,” Software Quality Journal, vol. 15, no. 4, pp. 365-381, 2007.
[185] C. Becker, D. Walker, and C. McCord, “Intertemporal Choice: Decision Making and Time in Software Engineering,” 2017 IEEE/ACM 10th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE), pp. 23-29, 2017.
[186] W. Cunningham, “The WyCash portfolio management system, in: 7th International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA ’92),” pp. 29–30, 1992.
[187] H. Ghanbari, T. Besker, A. Martini, and J. Bosch, “Looking for Peace of Mind? Manage your (Technical) Debt - An Exploratory Field Study,” 11th International Symposium On Empirical Engineering and Measurement (ESEM), 2017.
[188] N. Brown et al., “Managing technical debt in software-reliant systems,” Proceedings of the FSE/SDP workshop on Future of software engineering research, pp. 47-52, 2010.
[189] F. Fagerholm et al., “Performance Alignment Work: How software developers experience the continuous adaptation of team performance in Lean and Agile environments,” Information and Software Technology, vol. 64, pp. 132-147, 2015.
[190] P. Ralph, and P. Kelly, “The dimensions of software engineering success,” Proceedings of the 36th International Conference on Software Engineering (ICSE), pp. 24-35, 2014.
[191] S. Beecham, N. Baddoo, T. Hall, H. Robinson, and H. Sharp, “Motivation in Software Engineering: A systematic literature review,” Information and Software Technology, vol. 50, no. 9–10, pp. 860-878, 2008.
[192] J. M. Verner, M. A. Babar, N. Cerpa, T. Hall, and S. Beecham, “Factors that motivate software engineering teams: A four country empirical study,” Journal of Systems and Software, vol. 92, pp. 115-127, 2014.
[193] D. Laribee, “Code Cleanup - Using Agile Techniques to Pay Back Technical Debt,” Msdn, no. 2019-10-07, pp. https://msdn.microsoft.com/en-us/magazine/ee819135.aspx, 2009.

المنارة للاستشارات

www.r

[194] B. Vogel-Heuser, and E.-M. Neumann, “Adapting the concept of technical debt to software of automated Production Systems focusing on fault handling, mode of operation and safety aspects,” IFAC-Papers OnLine, vol. 50, no. 1, pp. 5887-5894, 2017.
[195] J. Keyes, “Social software engineering,” Auerbach Series, 2011.
[196] R. Alfayez, P. Behnamghader, K. Srisopha, and B. Boehm, “An Exploratory Study on the Influence of Developers in Technical Debt,” IEEE/ACM International Conference on Technical Debt (TechDebt), pp. 1-10, 2018.
[197] M. J. Salamea, and C. Farré, “Influence of Developer Factors on Code Quality: A Data Study,” 19th International Conference on Software Quality, Reliability and Security Companion (QRS-C), pp. 120-125, 2019.
[198] B. Hardy, “Morale: definitions, dimensions and measurement,” PhD diss. University of Cambridge, 2010.
[199] A. C. C. França, T. B. Gouveia, P. C. F. Santos, C. A. Santana, and F. Q. B. d. Silva, “Motivation in software engineering: A systematic review update,” 15th Annual Conference on Evaluation & Assessment in Software Engineering (EASE 2011), pp. 154-163, 2011.
[200] S. McConnell. "Technical Debt. 10x Software Development [cited 2010 June 14]," 2019-10-07; http://www.construx.com/10x_Software_Development/Technical_Debt/.
[201] L. Peters, “Technical Debt: The Ultimate Antipattern - The Biggest Costs May Be hidden, Widespread, and Long Term,” Managing Technical Debt (MTD), 2014 Sixth International Workshop on, pp. 8-10, 2014.
[202] M. Lavallée, and P. N. Robillard, “The impacts of software process improvement on developers: A systematic review,” 2012 34th International Conference on Software Engineering (ICSE), pp. 113-122, 2012.
[203] C. B. Jaktman, “The influence of organisational factors on the success and quality of a product-line architecture,” Australian Software Engineering Conference (Cat. No.98EX233), pp. 2-11, 1998.
[204] G. Suryanarayana, G. Samarthyam, and T. Sharma, "Chapter 1 - Technical Debt," Refactoring for Software Design Smells, G. Suryanarayana, G. Samarthyam and T. Sharma, eds., pp. 1-7, Boston: Morgan Kaufmann, 2015.
[205] C. F. Evans Data Corp, and Stripe research, “The developer coefficient software engineering efficiency and its $3 trillion impact on global gdp. ,” https://stripe.com/files/reports/the-developer-coefficient.pdf, 2018.
[206] I. Ozkaya, “The Voice of the Developer,” IEEE Software, vol. 36, no. 5, pp. 3-5, 2019.
[207] A. Aldaeej, Towards Effective Technical Debt Decision Making in Software Startups: Association for Computing Machinery, 2019.
[208] R. H. Rasch, and H. L. Tosi, “Factors Affecting Software Developers' Performance: An Integrated Approach,” MIS Quarterly, vol. 16, no. 3, pp. 395-413, 1992.
[209] H. Alahyari, T. Gorschek, and R. Berntsson Svensson, “An exploratory study of waste in software development organizations using agile or lean approaches: A multiple case study at 14 organizations,” Information and Software Technology, vol. 105, pp. 78-94, 2019.
[210] D. Graziotin, F. Fagerholm, X. Wang, and P. Abrahamsson, “What happens when software developers are (un)happy,” Journal of Systems and Software, vol. 140, pp. 32-47, 2018.

المنارة للاستشارات

www.r

[211] T. Sedano, P. Ralph, and C. e. Péraire, “Software development waste,” In: Proceedings of the 39th International Conference on Software Engineering, Buenos Aires, Argentina, 2017, pp. 130-140.
[212] T. Besker, A. Martini, and J. Bosch, “Software developer productivity loss due to technical debt—A replication and extension study examining developers’ development work,” Journal of Systems and Software, vol. 156, pp. 41-61, 2019.
[213] C. Maslach, Schaufeli, W. B., & Leiter, M. P., “Job burnout,” In: Annual Review of Psychology, 52, 397-422. doi:10.1146/annurev.psych.52.1.397, 2001.
[214] V. Braun, and V. Clarke, “Using thematic analysis in psychology,” Qualitative research in psychology, 3(2), pp. 77-101, 2006.
[215] M. Vaismoradi, H. Turunen, and T. Bondas, “Content analysis and thematic analysis: implications for conducting a qualitative descriptive study,” Nursing & Health Sciences 15(3), pp. 398-405, 2013.
[216] H. Wickham, "ggplot2: Elegant Graphics for Data Analysis. Springer-Verlag New York," Springer Publishing Company, Incorporated, 2009.
[217] H. Ghanbari, T. Vartiainen, and M. Siponen, “Omission of quality software development practices: A systematic literature review,” ACM Computing Surveys, vol. 51, no. 2, 2018.
[218] R. C. Barnett, R. T. Brennan, and K. C. Gareis, “A closer look at the measurement of burnout,” Journal of Applied Biobehavioral Research, vol. 4, no. 2, pp. 65-78, 1999.
[219] Y. Guo et al., “Tracking technical debt - An exploratory case study,” In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), 2011, pp. 528-531.
[220] C. Seaman, “Using Technical Debt Data in Decision Making: Potential Decision Approaches,” Managing Technical Debt (MTD), 2012 Third International Workshop, pp. 45-48, 2012.
[221] S. McConell, "Managing Technical Det presentation at ICSE 2013," 2013.
[222] U. Flick, An introduction to Qualitative Research, 2009.
[223] D. Moitra, "Managing Organizational Change for Software Process Improvement," Software Process Modeling, S. T. Acuña and N. Juristo, eds., pp. 163-185, Boston, MA: Springer US, 2005.
[224] N. Zazworka et al., “A case study on effectively identifying technical debt,” In: Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering, Porto de Galinhas, Brazil, 2013, pp. 42-47.
[225] J. Yli-Huumo, A. Maglyas, and K. Smolander, “How do software development teams manage technical debt? – An empirical study,” Journal of Systems and Software, vol. 120, no. Supplement C, pp. 195-218, 2016.
[226] J. Holvitie, and V. Leppanen, “DebtFlag: Technical debt management with a development environment integrated tool,” In: Managing Technical Debt (MTD), 2013 4th International Workshop on, 2013, pp. 20-27.
[227] F. A. Fontana, R. Roveda, and M. Zanoni, “Tool support for evaluating architectural debt of an existing system: an experience report,” In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, Pisa, Italy, 2016, pp. 1347-1349.
[228] A. Martini, and J. Bosch, “The magnificent seven: towards a systematic estimation of technical debt interest,” In: Proceedings of the XP2017 Scientific Workshops, Cologne, Germany, 2017, pp. 1-5.

المنارة للاستشارات

www.r

[229] Q. Huang, E. Shihab, X. Xia, D. Lo, and S. Li., “Identifying self-admitted technical debt in open source projects using text mining,” Empirical Software Engineering. 1-34. Research Collection School Of Information Systems, 2017.
[230] M. Waseem, and N. Ikram, "Architecting activities evolution and emergence in agile software development: An empirical investigation initial research proposal," Lecture Notes in Business Information Processing, 2016.
[231] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, “Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt,” SIGSOFT Softw. Eng. Notes, vol. 38, no. 5, pp. 51-54, 2013.
[232] M. Chicote, “Startups and Technical Debt: Managing Technical Debt with Visual Thinking,” In: 2017 IEEE/ACM 1st International Workshop on Software Engineering for Startups (SoftStart), 2017, pp. 10-11.
[233] M. Unterkalmsteiner et al., “Software Startups – A Research Agenda,” e-Informatica Software Engineering Journal, vol. 10, no. 1, pp. 89–123, 2016.
[234] S. M. Sutton, “The role of process in software start-up,” IEEE Software, vol. 17, no. 4, pp. 33-39, 2000.
[235] M. Crowne, “Why software product startups fail and what to do about it. Evolution of software product development in startup companies,” In: IEEE International Engineering Management Conference, 2002, pp. 338-343 vol.1.
[236] S. Gralha, D. Damian, A. Wasserman, M. Goulao, and J. Araujo, “The Evolution of Requirements Practices in Software Startups,” In: International Conference on Software Engineering (ICSE), to appear, 2018.
[237] J. Yli-Huumo, T. Rissanen, A. Maglyas, K. Smolander, and L.-M. Sainio, “The Relationship Between Business Model Experimentation and Technical Debt,” In: Software Business, Cham, 2015, pp. 17-29.
[238] "https://www.sonarqube.org/."
[239] "https://anacondebt.com/."
[240] P. A. Gompers, “Grandstanding in the venture capital industry,” Journal of Financial Economics, vol. 42, no. 1, pp. 133-156, 1996.
[241] A. Grossman, Postmortems from Game Developer: New York: Focal Press., 2003.
[242] G. K. Hanssen, T. Stålhane, and T. Myklebust, "What Is Safety-Critical Software?," SafeScrum® – Agile Development of Safety-Critical Software, G. K. Hanssen, T. Stålhane and T. Myklebust, eds., pp. 17-29, Cham: Springer International Publishing, 2018.
[243] F. Duncan, “At the Sharp End: developing and validating Safety Critical Software,” In: Achieving Systems Safety, London, 2012, pp. 225-235.
[244] D. Feitosa et al., “Design Approaches for Critical Embedded Systems: A Systematic Mapping Study,” In: Evaluation of Novel Approaches to Software Engineering, Cham, 2018, pp. 243-274.
[245] A. B. Bujok et al., “Approach to the development of a Unified Framework for Safety Critical Software Development,” Computer Standards & Interfaces, vol. 54, pp. 152-161, 2017.
[246] R. Kasauli, E. Knauss, B. Kanagwa, A. Nilsson, and G. Calikli, “Safety-Critical Systems and Agile Development: A Mapping Study,” In: 2018 44th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2018, pp. 470-477.

المنارة للاستشارات

www.r

[247] J. Pedersen Notander, M. Höst, and P. Runeson, “Challenges in Flexible Safety-Critical Software Development – An Industrial Qualitative Survey,” In: Product-Focused Software Process Improvement, Berlin, Heidelberg, 2013, pp. 283-297.
[248] H. Ghanbari, “Seeking Technical Debt in Critical Software Development Projects: An Exploratory Field Study,” In: Proceedings of the 2016 49th Hawaii International Conference on System Sciences (HICSS), 2016, pp. 5407-5416.
[249] D. S. Cruzes, and T. Dyba, “Recommended Steps for Thematic Synthesis in Software Engineering,” In: 2011 International Symposium on Empirical Software Engineering and Measurement, 2011, pp. 275-284.
[250] A. Sharma, M. Kumar, and S. Agarwal, “A Complete Survey on Software Architectural Styles and Patterns,” Procedia Computer Science, vol. 70, pp. 16-28, 2015.
[251] Z. Codabux, and B. J. Williams, “Technical Debt Prioritization Using Predictive Analytics,” In: 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C), 2016, pp. 704-706.
[252] D. Falessi, and A. Voegele, “Validating and prioritizing quality rules for managing technical debt: An industrial case study,” In: Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on, 2015, pp. 41-48.
[253] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis, “Estimating the breaking point for technical debt,” In: 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), 2015, pp. 53-56.
[254] W. Snipes, B. Robinson, G. Yuepu, and C. Seaman, “Defining the decision factors for managing defects: A technical debt perspective,” In: Managing Technical Debt (MTD), 2012 Third International Workshop on, 2012, pp. 54-60.
[255] R. Reboucas De Almeida, U. Kulesza, C. Treude, D. Cavalcanti Feitosa, and A. H. G. Lima, “Aligning technical debt prioritization with business objectives: A multiple-case study,” In: Proceedings - 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, 2018, pp. 655-664.
[256] S. L. Pfleeger, and B. A. Kitchenham, “Principles of survey research: part 1: turning lemons into lemonade,” SIGSOFT Softw. Eng. Notes, vol. 26, no. 6, pp. 16-18, 2001.
[257] G. M. Sullivan, and A. R. Artino, Jr., “Analyzing and interpreting data from likert-type scales,” Journal Of Graduate Medical Education, vol. 5, no. 4, pp. 541-542, 2013.
[258] T. Besker, A. Martini, and J. Bosch, “Time to Pay Up - Technical Debt from a Software Quality Perspective,” Proceedings of the 20th Ibero American Conference on Software Engineering (CibSE) @ ICSE17, 2017.
[259] T. Besker, A. Martini, and J. Bosch, “Technical debt triage in backlog management,” In: Proceedings of the Second International Conference on Technical Debt, Montreal, Quebec, Canada, 2019, pp. 13-22.
[260] S. M. U Gneezy, P Rey-Biel, “When and why incentives (don't) work to modify behavior,” Journal of Economic Perspectives 25 (4), vol. 1261, pp. 191-210, 2011.
[261] P. Milne, “Motivation, incentives and organisational culture,” J. Knowledge Management, vol. 11, pp. 28-38, 2007.
[262] S. Bala, and J. Mendling, “Monitoring the Software Development Process with Process Mining,” In: Business Modeling and Software Design, Cham, 2018, pp. 432-442.

المنارة للاستشارات

www.r

[263] M. J. Bateman, and T. D. Ludwig, "Managing Distribution Quality Through an Adapted Incentive Program with Tiered Goals and Feedback," Journal of Organizational Behavior Management, vol. 23, no. 1, pp. 33-55, 2004.

[264] F. S. F. Soares, G. S. d. A. Junior, and S. R. d. L. Meira, "Incentive Systems in Software Organizations," In: 2009 Fourth International Conference on Software Engineering Advances, 2009, pp. 93-99.

[265] C. A. Ramus, "Encouraging innovative environmental actions: what companies and managers must do," Journal of World Business, vol. 37, no. 2, pp. 151-164, 2002.

[266] W. Snipes, V. Augustine, A. R. Nair, and E. Murphy-Hill, "Towards recognizing and rewarding efficient developer work patterns," In: 2013 35th International Conference on Software Engineering (ICSE), 2013, pp. 1277-1280.

[267] Y. Wang, and M. Zhang, "Penalty policies in professional software development practice: a multi-method field study," In: 2010 ACM/IEEE 32nd International Conference on Software Engineering, 2010, pp. 39-47.

[268] T. Punter, M. Ciolkowski, B. Freimut, and I. John, "Conducting on-line surveys in software engineering," In: 2003 International Symposium on Empirical Software Engineering, 2003. ISESE 2003. Proceedings., 2003, pp. 80-88.

[269] A. Martini, V. Stray, and N. B. Moe, "Technical-, Social- and Process Debt in Large-Scale Agile: An Exploratory Case-Study," In: Agile Processes in Software Engineering and Extreme Programming – Workshops, Cham, 2019, pp. 112-119.

المنارة للاستشارات

www.r