Tampere University

Teemu Orava

TECHNICAL DEBT MANAGEMENT IN SMALL AND MEDIUM-SIZED ENTERPRISES

Information Technology
Master of Science Thesis
August 2019

ABSTRACT

Teemu Orava: Technical Debt Management in Small and Medium-sized Enterprises
Master of Science Thesis
Tampere University
Software Engineering
August 2019

The need to release our products under tough time constraints has required us to take shortcuts during the implementation of our products and to postpone the correct implementation, thereby accumulating Technical Debt.

In this work, we report the experience of a Finnish SME (Small and Medium-sized Enterprise) in managing Technical Debt (TD), investigating the most common types of TD they faced in the past, their causes, and their effects. The case company is a spin-off which sells one product. Its development was outsourced in the beginning and later continued with external developers.

We set up a focus group in the case-company, involving different roles. The results showed that the most significant TD in the company stems from disagreements with the supplier and lack of test automation. Specification and test TD are the most significant types of TD. Budget and time constraints were identified as the most potential root causes of TD.

TD occurs when time or budget is limited or the amount and content of work are not understood properly. However, not all postponed activities generated ”debt”. Sometimes the accumulation of TD helped meet deadlines without a major impact, while in other cases the cost for repaying the TD was much higher than the benefits. From this study, we found out that learning from customers, careful estimations and continuous improvement could be potential strategies to mitigate TD.

These strategies include iterative validation with customers, efficient communication with stakeholders, improvement of meta-cognition in estimations, and value orientation in budgeting and scheduling.

Keywords: Case Study, Empirical Study, Technical Debt, Small and Medium-Sized Enterprise

TIIVISTELMÄ

Teemu Orava: Teknisen velan hallitseminen pk-yrityksissä
Diplomityö
Tampereen yliopisto
Ohjelmistotuotanto
Elokuu 2019

Tarve julkaista tuotteitamme tiukalla aikataululla on vaatinut meitä käyttämään oikopolkuja tuotteidemme kehityksessä ja viivästyttää oikeellista toteutusta, mikä on kerryttänyt teknistä velkaa.

Tässä työssä raportoimme suomalaisen PK-yrityksen kokemuksia teknisen velan hallinnasta, tutkien heidän kohtaamansa teknisen velan yleisempiä tyyppejä, syitä ja vaikutuksia. Kohdeyritys on spin-off, joka myy yhtä tuotetta, jonka kehitys aluksi ulkoistettiin ja jota sittemmin jatkettiin ulkoisten kehittäjien kanssa.

Teimme kohdeyritykselle fokusryhmähaastattelun, johon osallistui eri rooleissa olevaa henkilöstöä. Tulokset näyttivät, että yleisin tekninen velka on seurannut erimielisyyksistä toimittajan kanssa ja testiautomaation puutteesta. Määritelmien ja testien tekniset velat ovat yleisimpiä teknisen velan tyyppejä. Budjetti- ja aikarajoitteet tunnistettiin potentiaalisimmiksi teknisen velan juurisyiksi.

Tekninen velka aiheutuu, kun aika tai budjetti on rajallinen tai kun työn määrää ja sisältöä ei ole ymmärretty oikein. Kaikki viivästetyt aktiviteetit eivät kuitenkaan luonneet “velkaa”. Joskus teknisen velan kertyminen auttoi kohtaamaan aikarajat ilman merkittäviä vaikutuksia, kun taas muissa tapauksissa teknisen velan takaisinmaksamisen kulu oli paljon suurempi kuin hyödyt. Tässä työssä havaitsimme, että asiakkailta oppiminen, varovaiset arviot ja jatkuva parantaminen ovat potentiaalisia strategioita teknisen velan vähentämiseen.

Nämä strategiat kattavat iteratiivisen validoinnin asiakkaan kanssa, tehokkaan viestinnän sidosryhmien kanssa, metakognition parantamisen arvioissa ja arvolähtöisyyden budjetoinnissa sekä aikataulutuksessa.

Avainsanat: Tapaustutkimus, Empiirinen tutkimus, Tekninen velka, PK-yritys

PREFACE

I want to thank my supervisor, Davide Taibi, for guiding me to the path of researching technical debt, the concept which I have been long aware and concerned, and learned to understand better and better during the process of writing. I want to thank our case company for providing their experiences to our knowledge. I want to thank all who contributed to my work. I want to thank my loved ones; my fiancée and my family who stood for me during my quest. I hope our work can make this important concept recognized even wider, resulting better quality, better usability, better management, better business and better software. May you wisen in the management of your technical debt.

In Tampere, 19th August 2019

Teemu Orava

CONTENTS

LIST OF TABLES

LIST OF SYMBOLS AND ABBREVIATIONS

ASD	Agile Software Development
ATAM	Architecture Trade-off Analysis Methodology
CFO	Chief Financial Officer
CMO	Chief Marketing Officer
CTO	Chief Technology Officer
LSD	Lean Software Development
MVP	Minimum Viable Product
OD	Organizational Debt
PoC	Proof of Concept
ROI	Return of Investment
RQ	Research Question
SME	Small and Medium-sized Enterprise
SUT	System Under Test
TD	Technical Debt
TDD	Test-Driven Development
TDM	Technical Debt Management
UI	User Interface
UX	User Experience
VOC	Voice of the Customer
WIP	Work in Progress

1 INTRODUCTION

Companies commonly spend time on activities that improve software quality. These activities include refactoring aimed at removing technical issues that are believed to impact software qualities. Many factors can lead to technical debt; they can be internal, related to the business or the environment, or they can be external to the company [51], [10].

Technical Debt (TD) is a metaphor from the economic domain that ”refers to different software maintenance activities that are postponed in favor of the development of new features in order to get short-term payoff” [17]. According to N. Brown et. al., TD is a gap between current state and ideal state where system would perform optimally in desired environment [14].

Technical issues include any kind of information that can be derived from the source code and from the software process, such as usage of specific patterns, compliance with coding or documentation conventions, architectural issues, and many others. For example, when a new feature does not fit the current architecture, the incompatibility might solved with an immature implementation [17] than will be fixed in the future implementing a proper solution.

Researchers have investigated different aspects of TD and its management (TDM), and proposed different approaches for repayment. However, only few works have investigated concrete cases and identified the root causes of Technical Debt in companies.

In this work, we report on an empirical study we performed in our company, a Finnish SME that operates in Business-to-Business sector and develops a web application for managing sales channels.

The application’s development started four years ago outsourced to a vendor. Users can manage leads of the whole organization and see statuses of all the leads at once. Significantly, the first version had no possibility to define products for the leads but it was later added because of the customer demand.

In the beginning, the development of the product was outsourced to an external supplier. The product was developed as a Minimum Viable Product (MVP) [42] which acted as a PoC (Proof of Concept). Startup companies can find building a MVP beneficial to validate their business idea. According to E. Ries, the author of lean startup, the product has to be pivoted when it matures meaning that it has to be modified to meet new requirements [60]. According to E. Klotins et. al. limited resources force startups to make technical

compromises [38]. An inflexible MVP can create TD and slow down the development.

We identified cases where we postponed different activities and then analyzed the reason(s) for the postponement, the issues generated by the postponement, and how the postponed activities were implemented later. We also highlight the overhead generated by the postponement of the activities themselves (the interest).

The results of this work can be beneficial not only for the scientific community but also for other companies. As other companies can understand the reasons why we postponed some activities, and the issues generated by the postponement, they can make more informed decisions in similar situations. The results of this work confirm that TD can cause significant economic losses if payback is postponed. Also, postponing activities - even if it is beneficial in the short term - can often be an economic disadvantage.

We investigated our company's TD with a focus group involving five members of the company. Our main goal was not to regret past losses, but to understand what happened in the past and find ways to prevent similar situations.

The remainder of this paper is structured as follows. Section 2 reports on related work. In Section 3, we introduce the empirical study design and report the results in Section 4. The discussion is presented in Section 6 and conclusions are drawn in Section 7.

2 RELATED WORK

Related work in software and management fields is explored in light of TD. Its usefulness for its part in mitigating TD is estimated later in Discussion, section 6.

2.1 Technical debt

W. Cunningham introduced the concept, later to be known as technical debt, in his article from 1992 by stating "shipping first time code is like going into debt". He claimed the motivation to take TD by stating "a little debt speeds development so long as it is paid back promptly with a rewrite". The risk of TD lies in the postponement of repayment, which increases the interest of debt. TD bankruptcy can be considered as situation where "unconsolidated implementation" has brought the whole organization "to a stand-still". He saw waterfall development cycle, "working out a program in detail before programming begins", as a cure to software crisis, however questioned how it fits to object-oriented programming. He stated that polymorphism with objects had allowed flexibility for changes. [17]

On every epoch, there has been faith on the help of current trends. However, as stated by F. Brooks in 1986, there still does not exist a silver bullet; "there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity" [13]. Technology push and market pull has generally exceeded the supply of well-managed, high-quality software. Answering to demand may require taking TD, and uncontrolled accumulation of TD creates risks. Thus, TD has to be identified and communicated throughout the organization. The nature of future risks cannot be predicted, whereupon technical debt management requires risk management and acknowledgement of the existence of unknown through e.g. cost-benefit-analysis, and cause and effect analysis [69] [30].

2.1.1 Motivation

According to Z. Li et. al., technical debt occurs when technical shortcuts are taken to gain short-term benefits that are harmful for the system in the long term [49]. There are several reasons that lead to technical compromises, such as unrealistic schedule, budget constraints, or estimation errors. Highly indebted products become inflexible and

unprofitable, and the accumulation of debt eventually leads to technical debt bankruptcy whereupon the system has to be replaced with a new one.

E. Tom et. al. found out in their exploratory case study that reasons to take TD are pragmatism, prioritization, processes, attitudes, ignorance and oversight. *Pragmatism* is typical for small companies that implement MVP to conquer the markets and taking TD is only way to survive. In *prioritization*, insufficient budget is given to maintenance, because its value is not seen. For instance, if a PoC or a throwaway prototype, that was not intended to be scaled for production, is adopted as working product, it calls for taking TD. Lack of *processes* can hide TD and increase risk of taking it unknowingly. Code reviews decrease risk of accidental TD, because they encourage to internal audit and self-inspection. A good process will aim to identify and mitigate TD. Excessive complexity will be avoided if it needs to be explained. *Attitude* can have positive or negative impact to TD. Developer's fear can prevent an attempt to repay TD. However, their recklessness can lead to taking TD. *Ignorance* and *oversight* are sources of inadvertent debt, which consequences are not understood [27]. Oversight might prevent developer to see the future needs. Ignorance, lack of domain knowledge or technical knowledge, can lead to poor decisions. [73]

According to E. Tom. et. al., bankruptcy happens when system is not scalable or flexible enough, and rewrite takes less time than improving existing code. Tactical, incremental and inadvertent debt, which is not under control, is most likely to lead to bankruptcy. [73] According to systematic literature review by A. Ampatzoglou et. al., bankruptcy happens when project is cancelled or completely rewritten due to large maintenance costs caused by accumulated TD [4].

Klinger et al. [37] interviewed four software architects to understand how decision-making regarding TD was conducted in an enterprise environment. The results showed that the decisions related to TD issues were often informal and ad-hoc, which prevented tracking and quantifying the decisions and issues. Moreover, just as in our work, this study also reported that there was a large communication gap between technical and business stakeholders in the discussions related to TD.

Recently, De Toledo et al. [19] conducted an exploratory case study with a large company on a project with about 1,000 services. They investigated Architecture Technical Debt in the communication layer. The study combined an analysis of existing documentation and interviews to identify issues, solutions, and risks, providing a list of architectural issues that generate technical debt.

N. Ernst et. al. find architectural issues the most important cause for TD. Architectural issues are difficult to manage, because they might become submerged in long term. Furthermore, developers usually consider TD more critical than managers. [25] ATAM (Architectural Trade-off Analysis Methodology) [36] can be used to make optimal architectural decisions by hearing all stakeholders. Meanwhile, profitability of decisions has to be considered. [65]

In a complex software, modularity helps to handle the complexity. Modules should be loose-coupled to decrease dependencies. Modularity should be organized according to communication structure. [6] However, as suggested by K. Beck et. al. in Agile Manifesto, self-organized teams are most efficient [9]. To ease development's burden, organizations should ensure that developers are provided with proper facilities for development [9] not to have excessive cognitive load from meta-work. Architectural decisions should be done carefully. ATAM can help to find possible trade-offs in architecture decisions hearing formally all stakeholders [36].

One cause of TD related to specification issues is lack of validation. According to E. Ries, startups fail when they make plans too carefully while market is actually unpredictable. Instead of speculating they should go to the field and understand how to satisfy customer needs iteratively. In lean startup, validated learning and pivoting when needed helps to keep product up-to-date with customer needs. [60] Having its roots in Customer Development coined by S. Blank [11][12], goal is to develop what customer actually needs and to avoid Organizational Debt (OD) which accumulates when customer needs are not met. Customer Development differs from an approach which tries to predict customer need. Best information lies in interacting with customers. Validating can be integrated as part of development process. Similarly, in quality function deployment (QFD) introduced by Y. Akao, voice of the customer (VOC) is basis of the value chain in product development [1].

According to M. Christel et. al., issues that occur in requirements elicitation are problems of scope, problems of understanding and problems of volatility. Customers can be uncertain of their needs and technical boundaries. Consequently, requirements elicitation has to be started early and kept simple. [15]

Even with iterative validation, undesired TD can still occur. Estimation errors are one reason leading to accumulation of TD. Underestimation leaves little time and budget for the company to deliver its promises and can eventually force it to take TD. Dunning-Kruger effect can explain estimation errors. According to J. Kruger et. al., a small amount of knowledge leads to overconfidence of own competence which explains why complexity of tasks is underestimated [41]. Overconfidence caused by underestimation helps to proceed with the task but it's harmful when budgeting and scheduling are based on these estimations. As M. Fowler stated, development of every system requires learning [27]. In Dunning-Kruger effect, the meta-cognition of own competence is faulty and it improves as experience of the task increases. Until then, skepticism against the accuracy of estimations helps.

Optimism bias and wishful thinking can lead to taking TD. According to E. Allman, Murphy's law should be taken in account on taking TD: "anything that can go wrong, will go wrong". Developers are rewarded for avoiding TD but for fast delivery. Developer is often the one who takes or is forced to take TD. He might be the only one who is aware of TD and consequences. However, developer might not be the one who repays interest. At latest, consequences of TD might only became evident in customer loss. TD management requires proactivity, and repayment has to be pre-scheduled. Furthermore, in the

beginning, the team seldom has comprehends the problem fully. Waterfall model expects sequential development to finalize requirements, design and development in order. However, the consequences of TD will be emphasized because changing of requirements is inevitable during the sequential process, which hopes to freeze requirements that are unstoppable. Thus, he suggests an iterative, agile process which plans for change, and a working prototype is used for customer tests continuously. [2]

Allman states that TD affects all stakeholders in different ways; customers, help desk, operations, engineers, marketing and management. *Customers* seem the ones who force to take TD but they also suffer the consequences when usability decreases. However, they don't have control on TD. Thus, customer should be advised about the risks of TD when deciding about budget and schedule. Nor does *help desk* have control on TD but they are the first to interact with the customer when problems occur, for instance caused by TD. Pressure to help desk from customers will even increase by time, and time to fix the problem is increased by accumulation of TD. *Operations* as well suffer from the consequences of TD without having a possibility to affect to its causes. However, DevOps has increased the cooperation between development and operation. Thus, operation's perspective and long-term maintainability has become more considerable factor. *Engineers* include developers and maintainers. Developers who implement all the functionality as soon as possibly are often favored over developers who also see from the perspective of long-term maintenance and take possibly longer time while developing maintainable, reliable code. Consequences of intentional TD may not be realized, and unintentional TD is not even recognized; "in its early days technical debt is almost invisible, because the interest payments haven't started coming due yet." Thus, developers who are also experienced maintainers are most reliable in TDM. [2]

Furthermore, *marketing*, pressured by sales, often give promises to customers of fast delivery and maximum functionality. However, when proven impossible by delays and the decrease of quality, they also need to negotiate with the customer. Bug fixes for their part steal time from new functionality when failure demand surpasses value demand. Marketing at its best can also communicate quality. *Management* can be sub-optimizing if it favors certain departments. If marketing is favored over others, management goes along marketing, and tries to increase sales and thus increase TD. When TD accumulates, management has to survive from impacts of TD to sales and public relations, when it decreases quality and increases response times. Allman states, "good management understands risk management and balances out the demands of all departments". Management can understand technical debt in analogy of financial debt. Technical people can help others to understand consequences of "mismanaging" TD. [2]

2.1.2 Types

Ampatzoglou et al. [3] conducted a multiple case study in the embedded systems industry in order to investigate the expected lifetime of components affected by TD and the most

frequently occurring types of TD. They considered seven embedded systems industries from five different countries. The results showed that in order to increase the expected lifetime of components, maintainability plays a major role. Moreover, they found the most frequent types of TD to be test, architecture, and code.

Documentation is a common type of TD, because it gets easily compromised on account of more visible work that brings more direct value to customer. Documentation keeps software understandable between developers as complexity increases. A complex software can be impossible to comprehend fully. Documentation removes bottleneck caused by communication. Documentation reduces tacit knowledge and thus, it decreases the risk to lose knowledge.

Documentation can form the foundation for software and help developers to hold consensus. Documentation doesn't have to be all-inclusive as it may become out-of-date. Like mentioned in Agile Manifest, working software has higher importance than comprehensive documentation [9]. At simplest, documentation can consist of code comments describing each class method. Design by contract (DbC) expects compatibility of classes. It also allows to prioritize on conforming behavior instead of increasing complexity to prepare for non-occurring situations. In opposite to DbC, in defensive design, components prevent violations of preconditions which can improve safety in API (Application Programming Interface).

According to M. Poppendieck [57], automated tests also act as documentation as they communicate the intention of system's functionality unambiguously, give feedback of the system's health to prove the integrity and provide scaffolding for refactoring and changes in the last responsible moment.

S. McConnell separates TD to 2 main types, intentional and unintentional TD. Intentional TD is taken for short-term benefit and is under control. Unintentional TD is caused by low quality and is harmful, because its consequences are not known. Intentional TD can be divided to short-term and long-term TD. Short-term TD is reactive and tactical while long-term TD is proactive and strategical. Short-term TD can be focused or unfocused. Focused TD is trackable and manageable while unfocused TD is difficult to identify and manage. Focused TD consists noticeable larger-scale decisions while unfocused TD are smaller-scale decisions which can exist as hundreds of shortcuts in the code. Incomplete work that doesn't require interest payment is not debt. Postponed, cancelled or in-complete features should not be considered as TD. [52]

According to M. Fowler, even the best teams deal with TD, but they also understand their consequences. He identifies 4 TD types. *Prudent debt* does not have severe consequences. *Deliberate debt* means that team understands its consequences while *reckless debt* means that team does not care about the consequences. *Inadvertent debt* means that team does not understand its consequences. Since most teams deal with prudent and deliberate debt, taking reckless and inadvertent debt on top of them is unsustainable. Only debt that yields short term benefit should be taken and repaid as soon as possible.

He claims that development of every system requires learning and thus, optimal solution is seldom reached on the first attempt. [27]

2.1.3 Impacts

According to E. Tom et. al., TD impacts to team morale, productivity, product quality and project risk. Developers, who take pride of their code, might find their *morale* deteriorate when they end up in a vicious cycle of TD, where they are given no time to repay previous TD and will become careless of the consequences. TD hinders maintenance and kills *productivity*. TD is financially analogous, and as its interest it will lower team velocity in long-term for short term acceleration. Immediate value is important especially for startups. If TD repeatedly slows down productivity, it tempts team to take more TD when new deadlines approach, increasing risk of TD bankruptcy. TD impacts on *product quality* and its decline might become noticeable for customers. TD obfuscates intention of code whereupon defects will be more difficult to notice. Quick solutions fail to support integrity. Unnoticeable TD such as inadvertent debt and incremental debt are most likely to increase risk. Solution is to identify occurred debt whereupon unknown unknown becomes known unknown. However, as deadlines approach, fixing TD might introduce more *project risk*. Hence, time has to be found to mitigation of TD. [73] Consequently, proactivity over reactivity in TDM (Technical Debt Management) is favorable to avoid risks.

Difficulty in measuring monetary cost of TD provides a challenge to proactivity in TDM. It is because developer's workload caused by TD is difficult to predict. [73] M. Stochel et. al. suggest that TD's return on investment (ROI) and profitability should be measured to understand its consequences and help prioritization. Furthermore, to avoid suboptimization, TD's impacts has to be considered on all levels; design, architecture and portfolio. Value-basis provides bridge between business and engineering. They found out that TD's consequences are problems in maintenance and increased number of defects. Thus, TD value is good measure for future quality. [65]

T. Sedano et. al. conducted an empirical study for eight software companies to reveal wasteful activities that don't produce any value to customer. They revealed that TD can cause extraneous cognitive load which is considered as a waste. When TD accumulates, TD adds complexity of software and makes software harder to comprehend or modify, which slows down team's response time. When customer demands changes or new features, there is no time left to refactor TD. Development tasks require learning capability which will be handicapped by extraneous cognitive load caused by TD. Furthermore, approaching deadlines cause distress, also a waste, which can lead to making poor decisions, of which TD can be considered as an example. [62]

2.2 Technical Debt Management

Z. Li et. al. in their systematic mapping study collected 9 TDM (Technical Debt Management) activities: identification, measurement, prioritization, prevention, monitoring, repayment, representation and communication [49]. These activities are described in subsections below. Some of these activities are emphasized in our case study.

2.2.1 Identification

Especially unintentional TD requires to be identified to become visible. According to Z. Li et. al., code analysis is most studied TD identification approach. TD can also be identified with code analysis, dependency analysis, check list and solution comparison. In *code analysis*, code issues, lack of tests, and design or architecture issues implied by code are analyzed and identified. In *dependency analysis*, component dependencies are analyzed. In *check list*, predefined scenarios are utilized to identify TD. In *solution comparison*, actual solution is compared to optimal solution dimensionally using technique such as cost-benefit analysis, where the distance shows the criticality of TD. [49]

2.2.2 Measurement

According to Z. Li et. al., calculation model is most studied TD measurement approach. Criticality of TD can also be measured with code metrics, human estimation, cost categorization, operational metrics or solution comparison. In *calculation model*, TD is calculated using "mathematical formulas or models". In *code metrics*, metrics are applied to source code. In *human estimation*, expertise is utilized. In *cost categorization*, cost types of handling incurred TD are estimated. In *operational metrics*, TD is indicated by "quality metrics of product operation". *Solution comparison* is described above. [49]

2.2.3 Prioritization

Especially limited budget requires prioritizing repayment of TD. According to Z. Li et. al., prioritization approaches are cost-benefit analysis, high remediation cost first, portfolio approach and high interest first. In *cost-benefit analysis*, TD with highest cost-benefit ratio of repayment is repaid first. In *high remediation cost first*, TD with highest repayment cost is repaid first. In *portfolio approach*, repayment of TD, new features and bugs are compared as risks and assets, and collected as asset set, maximizing ROI (Return of Investment) and minimizing the investment risk [29]. In *high interest first*, TD items "incurring higher interest should be repaid first". [49]

2.2.4 Monitoring

Understanding accumulation of TD requires monitoring. TD can be monitored with threshold-based approach, TD propagation tracking, planned check, TD monitoring with quality attribute focus and TD plot. In *threshold-based approach*, thresholds for TD quality metrics are defined, issuing warnings when not met. In *TD propagation tracking*, influences in through dependencies between parts containing TD and other parts are tracked. In *planned check*, change of identified TD is regularly measured and tracked. In *TD monitoring with quality attribute focus*, changes of quality attributes that harm TD, such as stability, are monitored. In *TD plot*, temporal trends in aggregated measures of TD are observed. [49]

2.2.5 Repayment

According to Z. Li et. al., *refactoring* is most studied TD repayment approach. In *refactoring*, code, design and architecture is altered to improve internal quality while preserving external behavior. [49] According to M. Poppendieck, refactoring is not rework, because it avoids waste, provides business value to customers and increases team velocity. As customer needs change during life-cycle, software has to be designed to be easily refactored, and excessive complexity can be considered as TD. [57]

Other approaches mentioned are rewriting, automation, re-engineering, repackaging, bug fixing and fault tolerance. In *rewriting*, code containing TD is rewritten. In *automation*, manual work, e.g. manual tests, manual builds or manual deployment, are automated. In *reengineering*, existing software is evolved for new behavior, features or operational quality. In *repackaging*, cohesive modules with manageable dependencies are grouped to simplify the code. In *bug fixing*, known bugs are resolved. In *fault tolerance*, runtime expections are placed strategically to avoid TD. [49]

S. McConnell claims that technical debt is easier to pay back gradually than entirely at once. He states that technical debt should be paid back when team velocity and response time to emergency decreases, or when quality starts to suffer from TD and customers notice the shortcuts [52]. However, debt amnesty can be achieved if debt will become written out. Also, debt can retire at the end of system's life-cycle. Companies should focus on TD that doesn't require repayment. Shortcuts that accumulate in code by time are most likely to avoid repayment. Developers are less motivated to repay unfocused TD, because benefit is not as evident as in focused TD, that are larger decisions that have bigger impact. [73]

2.2.6 Communication

Communication of identified TD is important to increase the recognition in the organization, spanning to decision-making in management-level. According to Z. Li et. al., communication approaches are TD dashboard, backlog, dependency visualization, code metrics visualization, TD list and TD propagation visualization. In *TD dashboard*, TD items, types and amounts are displayed in dashboard to inform all stakeholders of their existence. In *backlog*, TD items are handled equally in backlog along with known bugs and WIP features. In *dependency visualization*, undesirable dependencies, e.g. complex dependencies of components are visualized. In *code metrics visualization*, low measured quality of software, e.g. code complexity is visualized. In *TD list*, identified TD items are listed and made visible for all stakeholders. In *TD propagation visualization*, connections between TD items are shown. [49]

2.2.7 Prevention

There are several ways to prevent a product becoming indebted. Z. Li et. al. mentioned development process improvement, architecture decision making support, life-cycle cost planning and human factors analysis. In *development process improvement*, process is improved to prevent TD from occurring. In *architecture decision making support*, architecture design options are chosen by lowest potential risk of TD. In *life-cycle cost planning*, cost-effective plans are developed to minimize overall life-cycle TD. In *human factors analysis*, unintentional TD caused by human factors, e.g. indifference and ignorance, is minimized. [49]

Furthermore, M. Fowler suggests that software should be designed to be *strangled*, i.e., to be surpassed by new versions easily [26], while according to Cunningham, utilizing the *modularity* of objects allows developing flexible software [17]. However, sometimes debt cannot be avoided and in order to avoid rising costs, the generated debt should be paid back as soon as possible.

K. Tate claims that in sustainable software development, cost of change is kept constant by keeping number of defects low. If defects accumulate, eventually cost of change will increase, team velocity will decrease, and ability to respond will collapse. Defect burden is caused by poor decisions and accumulation of TD. Principles are favored over practices which would not sustain the complexity. Development is kept sustainable by following four principles: continual refinement of practices, working product, continual investment in design and valuing defect prevention over defect detection. [70]

2.3 Project Management

According to Z. Li. et. al., TDM as part of project management is challenging because business value of TD is difficult to measure. Developers are more aware of TD's value and consequences. Arguing its significance to management is challenging. In agile development, scheduling both TD prevention and repayment is challenging. Decision making in TDM requires identifying, measuring, and formalizing cost and benefit of TD. [49] Next, we summarize management frameworks and their relation to TDM.

By following existing management frameworks, companies can become more efficient in getting work done and mitigating TD. Especially startups deal with uncertainty where requirements are not yet clear and they still have to deliver value. An iterative model is most beneficial for a company dealing with high uncertainty. According to D. Taibi et. al., companies without any process are slowest to deliver [67].

2.3.1 Kanban

Delivering value on time and removing wasteful work that doesn't bring value requires managed framework. As a solution, kanban was introduced in just-in-time production by T. Ohno [54]. He lists defects and over-processing in seven wastes of JIT [54]. As TD hinders development and can cause new defects, it can be considered as a waste [65]. According to T. Sedano, TD is waste, because it causes extraneous cognitive load. Furthermore, when refactoring it skipped because of approaching deadlines, it leads to rework, which is also waste. [62] In simplest usage of kanban in software development, statuses of tasks are being tracked with kanban cards in a shared system. Work burns down as tasks are being moved from to-do status to done. Kanban cards provide visibility of tasks and their statuses to all personnel.

Kanban is focuses on decreasing work in progress (WIP). It provides more relaxed framework than Scrum and may lower the lead time [64]. Compared to Scrum, where development is time-boxed, in kanban, the development, changes and delivery are continuous. It bases development on customer pull instead of schedule push [57]. According to D. Taibi et. al., combination of Scrum and kanban decrease the need for communication [67] which otherwise would be taken away from available resources. Kanban helps to compress the value stream and to deliver as fast as possible. Kanban cards should be based on user story provided by customer and prioritized according to customer need [57].

Unfinished work, i.e. work-in-progress (WIP) is considered as a waste which only has value until it is finished and delivered successfully. Thus, companies need to deliver fast to decrease risk of obsolescence. It requires breadth-first approach instead of depth-first, and concurrent development instead of sequential development. When developers consider options before diving into details, development won't create waste while require-

ments become clearer. With fast delivery, they will be able to decide as late as possible when there is enough concreteness for fact-based decisions which removes the need for speculation. [57]

Openness is critical work the business. Organizations should avoid tacit knowledge and extraneous cognitive load caused by inefficient issue tracking. Lack of communication of tasks and their statuses hinders work allocation. To improve visibility, kanban cards can be used also for issue tracking. This can be seen as a way to follow lean methodology. When a defect is noticed, it is not memorized but fixed or delegated immediately. Efficient tracking system works as an information radiator which provides everyone visibility to issues [57]. It eventually increases value demand, decreases failure demand and improves quality, because issues will not be forgotten and they are seen by everyone. [59]

2.3.2 Agile Software Development

Based on Agile Manifesto by K. Beck et. al., Agile software development (ASD) puts emphasis on iterative collaboration with customers instead of following well-defined processes and plans. Like Occam's razor [23], it focuses on simplicity to deliver only what customers need and diminish the entropy caused by changes. A small company might want to avoid excessive processes and instead focus on getting work done. However, as in Conway's law, results of work are always affected by management of work.

By emphasizing self-organized teams, ASD expects teams to collaborate and make decisions independently without management intervention [9]. Like W. Deming has stated, teams must collaborate to share information about development. When quality is integrated to product which is continuously improved, there's no need for measurements and management by objective. The whole system should be estimated for improvement instead of individual contribution. [20]

Scrum is used in ASD as a framework to manage development in sprints as time-boxed iterations whereupon it provides a well-controlled process. Retrospective after every sprint gives an opportunity to improve the development process based on team's learnings.

According to P. Kruchten et. al., ASD provides a framework to repay TD in iterations, but ASD is still likely to attract TD. In rapid development, decisions are not considered in long term and systematic testing might be missing. Thus, TD is more likely to increment. In waterfall model and Big Design Up Front, risks are better considered, however, future changes are not carried. In ASD, TD can be avoided by identifying it and thus, including only intentional, prudent or deliberate TD and excluding inadvertent and unintentional TD. [40]

As an ASD method, M. Poppendieck et. al. presented following 7 simple rules for Lean Software Development (LSD) based on lean principles [57]. In below, we list these rules and their benefits in mitigating TD.

First, *eliminate waste* means eliminating all processes that don't bring value to customer. All unfinished work is waste, because it doesn't bring value to customer until it satisfies the customer need. In software development, following 9 wastes can be identified closely to wastes (muda) of lean philosophy. First, *building the wrong feature or product* means that company is not focusing on tasks valued by customer. Second, *mismanaging the backlog* means that company fails in attempt to prioritize on most important tasks. Third, *rework* occurs when company has not build valid features in the first place. Fourth, *unnecessarily complex solutions* are made when company doesn't maintain simplicity. Fifth *extraneous cognitive load*, sixth *psychological distress* and seventh *waiting/multitasking* occur when team roles are not clear enough. Eighth, *knowledge loss* is caused by lack of documentation. Ninth, *ineffective communication* occurs because of lack of team collocation or ineffective communication media. [57]

Eliminating waste and bringing customer value as a permanent aim prevents TD from occurring. Customer value has to be understood and prioritized in whole organization. According to M. Poppendieck et. al., companies need to identify types of their demand. Failure demand is taken away from value demand. If companies need to focus on fixing failures reported by their customers, they are taking less time to bring new value to customers and thus, they become less competitive. [59] Therefore, amplifying learning is crucial.

Second, *amplifying learning* requires short learning cycles to get regular feedback which create valuable knowledge for development. This is achieved with iterative delivery. When developer works close to customer they maximize their understanding of customer need. Developers also need to be synchronized with each other. Integration in small batches and regular builds with smoke tests synchronize development in overall and avoid regression. In set-based development, communicating possible solutions and constraints to stakeholders eventually saves time. [57]

Theory of causation and effectuation introduced by S. Sarasvathy explains importance of team collaboration [61]. Companies can try to causate what they consider valuable, but then they fail to consider their available resources. Developers have best understanding of technical boundaries and are able to effectuate value based on the existing resources. [57]

Third, *decide as late as possible* means that decisions are based on facts instead of speculation. Late decisions are possible when system is designed as flexible for change. In comparison to sequential development, concurrent development follows breadth-first approach instead of depth-first. It allows to postpone costly decisions until system is mature enough to receive feedback from customers. Instead of trying to build it right the first time, deciding as late as possible will defer bulk of decisions and prevent cost escalation. [57]

Fourth, *deliver as fast as possible* leads to less work in progress and less risk. It is a way to tolerate variability in the process. Value stream is compressed when backlog is deter-

mined by market pull instead of schedule push. Backlog visibility enables self-direction for developers, increasing performance compared to intervention of micromanagement. As in theory of constraints, team's performance is measured by its responsivity to value demand, where a team spending its capacity to failure demand is inefficient. [57]

Fifth, *empowering team* leads to better team efficiency and better results. Managers can help the team by representing VOC. Team has the best information of development status and empowering them to make decisions will have the basis of best available knowledge. Responsibility is a source of job satisfaction [31]. Feeling of progress is a source of intrinsic motivation [71]. Empowering team in LSD differs from CMM (Capability Maturity Model) and CMMI (Capability Maturity Model Integration) in a way that it does not aim on planning and evaluating capability in difficultly standardized work, but instead bases improvement on learning by empowering the team. [57]

Sixth, *building integrity in* is required to have consistency between product functionality and customer's objectives. Perceived integrity, relevancy to customer, is reflected by integrity of information flow from customers to developers. In sequential development, information flow is cut, because customer requirements cannot accommodate nor predict perceived integrity as a whole. When technical personnel communicate with end users and have continuous visibility customer values, design decisions and trade-offs they do are more favorable to perceived integrity. Managers don't necessarily have best available knowledge [33]. Short iterations provide frequent feedback loops and avoid feedback gaps [33]. [57]

Conceptual integrity, cohesiveness of system's central concepts, requires architecture to be efficient and flexible for future changes in perceived integrity. Components that are loosely coupled but have high cohesion retain maintainability and flexibility [33]. Refactoring complex design is required to pay back TD. Refactoring avoids waste and helps to respond to customer need and maintain quality, to deliver value as fast as possible which is the most important goal. As in lean production, development of new features should wait until root causes of complexity and problems in internal integrity are fixed. [57]

Automated tests enable refactoring as they act as a documentation and prove integrity by providing feedback of system health. Developer tests, consisting of unit, system and integration tests maintain conceptual integrity while customer tests maintain perceived integrity. [57]

Finally, seventh rule, *seeing the whole* is required to serve common good of the system. Sub-optimization decreases overall performance. It occurs because the real cause and effect is not seen or because factors that do not necessarily influence to success are being measured [5]. When management wants to measure performance in unstructured work that can be measured only partially or not at all, it leads to sub-optimization and decisions based on false information. [57]

When performance measurements are replaced with information measurements, attention is shifted from individuals to root causes of the defects. Performance measurements

shift the burden by focusing on symptoms and discourage cooperation. [57] Instead, P. Senge recommends following T. Ohno's Five Whys [54] to find root causes [63]. According to W. Deming, 20% of quality defects are rooted to employees while 80% are rooted to system [20]. Focusing on root causes of the defects helps to remove the obstacles of development, and to mitigate TD.

2.3.3 Contracts

Outsourcing can help companies to reach their goals on time while focusing on their core competence. Contract form is critical to build trust between parties and to make them work towards mutual benefit. According to M. Poppendieck et. al. [57] and K. Beck [8], an optional-scope contract will be most likely to match required work to the schedule and budget. It gives both sides an incentive to keep in the target-cost and target-schedule by limiting the scope. It differs from fixed-price contract in the sense that the vendor will not attempt to do less than required while keeping the fixed price. It differs from time-and-materials contract in the sense that the vendor will not attempt to do more than required, vice versa.

According to M. Poppendieck et. al. [57] and F. Thompson [72], when vendors bid a fixed-price contract, the lowest bid tends to come from the vendor which either does not understand the complexity or tries to win. The lowest bid is also the most tempting one for the customer, but it often leads to the situation where the vendor fails to deliver what was required. Expenses increase as the vendor needs to do the changes. An optional-scope contract encourages vendor to understand the problem in detail, decreasing the risk of TD.

2.3.4 Continuous Improvement

Continuous improvement of quality can follow PDSA cycle (Plan, Do, Study, Act) popularized by W. Deming. It increases standard iteratively by *planning* the process and its objectives, *doing* the changes, *studying* and examining the results, and *acting* and improving the process for next iteration. [21] Frequent feedback from customers and status updates from developers act as the source for the process.

According to D. Wood and Y. Yakup et. al. organizations tend to measure only the tip of the iceberg of quality costs while most significant costs are less visible and lie under the surface [78][79]. Thus, understanding TD in depth helps to mitigate it.

Mitigating TD can follow quality improvement models. There are competing views for quality improvement. Traditional model suggests that quality costs increase as level of conformity increases. According to C. Ittner, in continuous improvement quality costs eventually decrease as conformity increases because of learning opportunity. Traditional

model can be viewed as a static cost model while continuous improvement model is a dynamic model. [34]

Traditional model is useful for static, short-term analysis. Costs may increase temporarily but dynamic model helps to see quality improvement as an investment which pays back as increased knowledge and routine. Continuous improvement supports organizational learning and lowers the learning curve [81]. Continuous improvement provides a learning opportunity to developers to find better, project-specific design approach, since programming is learning as stated by M. Fowler [27].

C. Ittner mentions opportunity costs of poor quality as motivation for continuous improvement. Consequences of defects are farther than local. Continuous improvement ensures customer satisfaction which is risked if quality is not maintained. Dissatisfactory creates stronger Word of Mouth (WOM) than satisfactory, harming the business. In addition, he mentions indirect impacts of internal failure costs to productivity and variability, both affecting to customer response times and customer satisfaction. [34]

3 FOCUS GROUP

In this section, we describe the design of our study, including the goal, the research questions, the study context and procedure, and the data analysis.

The case company is a spin-off micro-enterprise. It has less than 10 employees and less than 2 million turnover, and falls into category of SME (Small and Medium-sized Enterprises). The case product is company's only product, a sales channel management tool offered as a SaaS (Software as a Service). The product has been developed for 4 years starting from January 2015. It was first outsourced to a vendor and later developed within the company. It is based on JavaScript and NoSQL and it is developed to MEAN stack (MongoDB, Express.js, AngularJS and node.js).

The product is a web application for managing sales channels. User can add products to which he can add sales leads. Users with different roles can be assigned to leads. Leads have contact information, contact personnel, question forms, statuses, reports, deadline dates and appointment dates. Products have KPI's (Key Performance Indicators) which can be controlled with quantitative questions. All lead data can be viewed in a grid. Leads are shown as dots in map in dashboard. Chosen KPI question controls the size of dot. Lead status controls the color of dot. This way the whole organization can see statuses of sales leads and have visibility to company's business. The version developed by supplier could have lead only for one product and it had to be copied to others. Later, external developers finished a feature where leads can be in multiple products.

3.1 Research Questions

We formulated the goal as follows, using the Goal/Question/Metric (GQM) template [7]:

Purpose	Analyze
Issue	occurrence of
Object	technical debt
Viewpoint	from the company's viewpoint

Based on the aforementioned goal, we derived the following Research Questions (RQs):

- **RQ1**: What are the most common types of TD?

- **RQ2**: What are the main causes of the accumulated TD?
- **RQ3**: How to mitigate TD?

RQ1 aims to determine the most common types of TD in the company and their impact on business. Its purpose is to identify the existing TD items and their criticality for the business.

RQ2 aims to investigate the causes of the TD identified in the company. It also intends to discover the root causes of TD in the company. The RQ's purpose is to understand reasons for the occurrence of TD.

RQ3 aims to identify ways to prevent TD from occurring in the future based on the knowledge gained by RQ1.

3.2 Planning the Study

As suggested by Kontio et al. [39], we planned a focus group to last from two to three hours. We identified a number of issues to be covered that were sufficient for having a meaningful discussion and interaction between the participants.

We selected five participants: the Chief Technology Officer (CTO), the Chief Financial Officer (CFO), the Chief Marketing Officer (CMO), and two developers. Different responsibilities of the participants provides a wide perspective and enables multifaceted discussion. All participants voluntarily participated in the study, as they were interested in how to avoid facing similar situations as in the past and wanted to understand which activities should not be postponed.

The session was moderated by one of the authors. Before the session, the moderator introduced the goals and the rules of the focus group. Then he presented the following six discussion topics:

T1: Which activities have been postponed in the past?
This topic was investigated in two steps: First, the participants answered this question individually, reporting the activities on post-it notes. Then the moderator asked them to read their list of activities and grouped the same activities on the whiteboard.

T2: Which type of Technical Debt was generated by the postponed activities?
The participants grouped the postponed activities based on the type of TD. We adopted a classification of eleven categories.

As our TD identification approach we used a check list as in [16] [49]. To discuss TD comprehensively, we included ten TD categories proposed by Li et. al. [49] (Requirement TD, Architectural TD, Design TD, Code TD, Test TD, Build TD, Documentation TD, Infrastructure TD, Versioning TD, Defect TD). As a relevant detail, vendor lock-in was added to Infrastructure TD.

In addition, related to TD, we wanted to discuss organizational factors and managerial decisions guided by organizational and environmental forces. Thus, we added one new category (Organizational Debt) as proposed by S. Blank et. al. [12].

The complete list of TD categories is reported in Table 3.1.

Requirement TD refers to implementation's distance from requirements specification [24].

Architectural TD refers to compromises in internal quality caused by architectural decisions.

Design TD refers to technical shortcuts in the design.

Code TD refers to code that does not apply best coding practices.

Test TD refers to shortcuts in tests.

Build TD refers to issues in build process.

Documentation TD refers to documentation that is lacking or does not serve its purpose.

Infrastructure TD refers to sub-optimal configuration that hinders development.

Versioning TD refers to issues in versioning.

Defect TD refers to defects and bugs in the system. [49]

Organizational Debt refers to compromises done in the organization [12].

T3: What were the reasons for postponement?
Regarding this topic, the participants discussed the motivations for the postponement of the activities and then reported them on the activity post-it notes created in T1.

T4: How were the activities then implemented?
In this task, the participants reported the solutions adopted to implement the postponed activities and reported them on the activity post-it notes.

T5: What problems did the postponement cause?
The participants collaboratively discussed the problems that caused the postponement, including economic, technical, and organizational ones. In this case as well, they reported them on the activity post-it notes.

T6: Ranking the importance of the problems.
In this task, each participant received ten votes, in the form of adhesive ”dots”, and was asked to vote on the most harmful problems. The participants were free to distribute their votes as they liked, for example, assigning all ten votes only to one activity or distributing them evenly among the activities.

Except for Topic 1, the participants were not limited to using only one post-it note per activity. If needed, the moderator allowed them to use extra post-it notes connected to

the same activity.

3.3 Data Analysis

The causes of TD were examined in order to better understand the causality of TD and to prevent TD from recurring. We followed the technique of the 5 Whys introduced by Ohno [54].

Table 3.1. *Table of TD types and their details*

TD Type	**TD Detail**
Requirements TD	Over-engineering
	Specification issues
	Other, please specify
Architectural TD	Architecture smells
	Architectural anti-patterns
	Complex architectural behavioral dependencies
	Violations of good architectural patterns
	Architectural compliance issues
	System-level structure quality issues
	Other, please specify
Design TD	Code smells
	Complex classes or methods
	Grime
	Incomplete design specification
	Other, please specify
Code TD	Low-quality code
	Duplicate code
	Coding violations
	Complex code
	Other, please specify
Test TD	Low code coverage
	Deferring testing
	Lack of tests
	Lack of test automation
	Residual defects not found in tests
	Expensive tests
	Estimation errors in test effort plan
	Other, please specify
Build TD	Bad dependencies
	Manual build process
	Flawed automatic building
	Build visibility debt
	Other, please specify
Documentation TD	Out-of-date documentation
	Incomplete documentation
	Insufficient documentation
	Lack of code comments
	Other, please specify

TD Type	**TD Detail**
Infrastructure TD	Old technology in use
	Old supporting tools in use
	Lack of continuous integration
	Lack of automated deployment
	Poor release planning
	Vendor lock-in
	Other, please specify
Versioning TD	Unnecessary code forks
	Multi-version support
	Other, please specify
Defect TD	Defects/bugs
Organizational Debt	Scheduling issues
	Communication issues
	Work allocation issues
	Prioritizing issues
	Budgeting issues
	Other, please specify

4 RESULTS

In this section, we will first report the perceived TD issues highlighted by the participants, together with the problems the issues generated and their causes. Finally, we will answer our research questions.

Personnel who participated the focus group were CFO (Chief Financial Officer), CMO (Chief Marketing Officer), CTO (Chief Technology Officer) and one developer. The focus group was conducted as following sessions:

- 5 minutes interview with CFO, CMO, CTO and one developer
- 2 hour 24 minutes interview with CFO and CTO
- 15 minutes interview with CTO and one developer

Input from CMO and one developer was only partial because they were not present in the longest session.

4.1 Perceived Debt

The participants if of focus group identified 10 different types of Technical Debt (TD items).

Organizational and Product Management Issues

TD1 *Product customization, as different customers wanted to use the system for different purposes. (Requirements TD, Organizational TD)* The prioritization of the features and tasks as well as the estimation of the cost and other effects of the customer-specific tailoring became difficult. The application originally did not identify lead's product which customer became to demand.

The feature became more expensive than thought as stated by CFO: "Developer came up with idea that no it's fine, we can do multiple products, it's not a big deal and the client paid us 14 000 for the customization and between our hours and what we paid to do that modification I think it costed like 150 000-170 000 euros to turn around. We accepted the specification but we totally did not understand how much would it actually cost and how much time it would take because it was done in a rush. [..] But thanks to that it was an investment in future and thanks to that we have been able to have other clients, some of the clients we have now because we

had made that change. Without that change we would not have been able to get those clients so eventually it was good but it was a big problem in our budget."

The recognized causes where Specification issues, Budget constraints, Estimation issues and Time constraints (e.g. related to Fast Delivery).

TD2 *Disagreement with supplier about the Minimum Viable Product (MVP). (Requirements TD)* The first version of the system was subcontracted from an external vendor that wanted to implement the initially agreed specification instead of iterative development and adapting to improved understanding of the customer needs.

The company and the supplier did not agree about the approach as CTO stated "basically they wanted to do this minimum viable product so that we could test the market and so on. It was fine but still asking that why actually do the architecture of the application in such manner. Even let say that if the answer is yes then you would have to change it afterwards basically much to get this vertical merging done. It does not still at least entirely clarify for me that they chose to do system architecture most probably the easiest way but if it means that then we get good results then we have to do from many parts all over again coding-wise. Even that we are proofing the concept I think the structure should be something that it's then for the future. So in this sense I think I'm in opinion still they should have listened to our point of view of having this exact thing differently then."

The company was aware of poor extensibility of MVP as CTO stated "MVP is actually most visible when it comes to these use cases and functionality that you have certain functionality in its easiest and minimum way that you can actually achieve the end result wanted. But I don't think that MVP should be taken into question when you actually do the foundation of the system. That should be still done so that you can actually build whatever on top of it."

The supplier did not provide a feature, that the company considered important, to MVP version. Multiple products were implemented after it was requested, but in addition, the system was supposed to allow leads to have interchangeable products. CFO stated, "the supplier did what they wanted to do and not what we needed them to do and that has created the problem why we don't have the vertical merging [the feature] because if they would have started with this vertical merging which we needed from them we wouldn't have to do it now three years and a half later. [..] That was why we were so upset with them because the plan was to have something not so solid in the back end but we could have a couple of customers to actually test. Problem is that they chose not to give us that we had to wait two years before to able to have a customer to test MVP and that was their big mistake. In that sense you're right, it really penalized us."

CTO stated, "I remember couple of tight discussions that it was a little bit like that it didn't go anywhere that basically we had our own view and they had their view."

CFO also stated that there was a communication gap between the supplier, who

had technical knowledge but lacked domain knowledge, and the company, who had domain knowledge but lacked technical knowledge: "I think that they kind of a undermined the views and what we needed. It was a little undermined by them because we were not knowledgeable of the coding world and the development world so basically they understood what they understood although they did not know anything about our world and what the clients needed. But because they are the ones with the technical knowledge they were thinking that we do it this way because that's the way it has to be done. I don't think they took that really much time in understanding because in every meeting we repeated the same. It was very important and in the specification, written specification and even in the contract they signed this was written."

Developer who was working at the supplier stated "the team was not able to convince that and explain the idea really well. Reason is being that the domain knowledge, the deficiency on the supplier's side."

The recognized causes where Specification issues, Budget constraints and Estimation issues.

Architectural Issues

TD3 *Lack of multitenancy causes budgeting increase and lack of flexibility (Infrastructure TD).* The products are delivered as SaaS services, but server maintenance is currently outsourced and forces a totally separated installation for each customer. This raises the operation and infrastructure costs. Multitenancy was not originally the highest priority and then the need of introducing it is costly. CFO described the issue followingly:

"We don't have multitenancy and costs us more because if we would have multitenancy we could actually create new servers and everything ourselves. It would be easy. We could do it directly without interference of technicality. So that's why also the maintenance costs us a little bit more because they [the supplier] know that when we need to add or reduce a server they need to use their expensive time for technicality to actually bring us server up or down. If we would have multitenancy we would not have that cost. We could do it ourselves or even the customers could do it themselves easily."

The recognized cause was Budget constraints.

TD4 *Hard to maintain a simple User eXperience (UX) with the growth of functionalities. (Design TD)* The UX was designed by the supplier that did not want to redesign it anymore, creating issues in adding new features while maintaining a good user experience.

The issue is also reoccurring because of limited time to maintain UX, as stated by CFO "what comes to design we have incomplete design specifications and we have had from the beginning because it was more a guideline. After this initial thing any

modification we had to do even in the very beginning did not have updated design. I think it has not been a big problem because I think the main design was still very nice and user-friendly and all. But I think now it starts becoming a problem as we create new pages."

According to CFO, the incurring complexity of UX affects also to customers at worst: "if we think about the experience as well, not only the looks but even the experience as well to make it we would need maybe some redesign now because it starts to be very complex. [..] The biggest problem when the design is not well planned is that it's affecting the way the potential customers, all the customers see your product. So if they see your product, if the design is not good they have the feeling that the product is very complex and it's going to take them too much time to use it and to learn how to use it. So right away they are like 'No, too complicated. I don't want to buy'. So it's actually a huge problem to commercialize a product that does not have a proper design. To be honest, we are getting there that we have some people who are like 'whoa, it looks so complicated'. We have had that comment a few times which we did not have a couple of years ago. Everyone was like 'oh, it's so simple'."

The recognized cause was Time constraints.

Development and Testing issues

TD5 *Lack of automatic testing costs more in the future* (Infrastructure TD). The testing budget was too low to enable the creation of automatic testing during development since the company did not even have enough time to concentrate on fast delivery to the client, as stated below by CFO.

"Testing budget was too low to enable, to create a long side development automated testing. Then we had the time constraint. We needed to prioritize the delivery in a shorter time. When you create automated tests the long side development takes longer. [..] Testers don't understand the business case so it takes longer for them to create anyway any test scenarios. At the latest stage when we are going to do the automated testing which is very important anyway it's going to costs us quite a lot because we need to dig into the old code of the application so we need to go back."

Testers lacked domain knowledge which decreased efficiency of testing, as stated by CTO: "testers we have had from the Supplier don't understand what the context the software is being used in. They can test this kind of dummy things that okay, button click does not give you the result but then they don't understand what the end user wants from it. [..] It has lowered the efficiency of the testing in my opinion. Let's say it's efficient to the certain point where they cannot anymore actually know that is this behaving like it should or not."

The recognized cause were Time constraints.

TD6 *Testing is expensive. (Test TD)* The company lacked dedicated testers and had human resourcing challenges, as stated by CFO "when the coder is not dedicated

[in testing] it's less efficient because that increases the cost naturally".

The focus group was not able to find the actual cause of this TD.

TD7 *Low code coverage in tests causes risks in development and additional work. (Test TD)* Supplier's testers lacked domain knowledge and testing took longer than estimated. Budget was limited for tests. CTO stated that customer is ready to pay only for the application part visible to them and if less visible costs are included, customer won't buy.

The recognized causes were Estimation issues, Communication issue and Budget constraints.

Source Code Maturity Issues

TD8 *Lack of code documentation. (Documentation TD)* Developers have been too busy to create code documentation as new features has usually highest priorities. It has resulted tacit knowledge which is harmful in collaborative work, as stated by CTO: "we at least to some extent have this technical debt about not having the documentation when you [developer] came and for example we did these bug fixes during the autumn and we still do. Had there been these I think it would have been a little bit easier to you."

CFO stated "we are uncovering issues when we do the things the cheapest way possible and we have no budget to fix those. Let's say you [developers] would work full time you could actually take the time to uncover document [documentation]. You could fix yourselves that we have couple of hours per day to do this kind of administrative issues so we get something a little bit cleaner."

The recognized cause was Time constraints.

TD9 *Technical shortcuts (Code TD)* These TD items are present mostly due to lack of time as stated by CFO "[technical shortcuts still happen because of] lack of time and money. In this case it would be more lack of time at this point because Developer is so busy and there is kind of long deadline".

The recognized causes were Time constraints and Budget constraints.

TD10 *Duplicated code (Code TD)* Developers failed to follow DRY (Don't Repeat Yourself) principle and modularize the implementations. Instead they duplicated the code because they were in hurry. In some case, the company had no time to extend or generalize the existing code. The focus group was not able to find the actual cause of this TD.

Table 4.1. *Perceived TD by interviewees and total points*

TD Description	Points
TD2. Disagreement with supplier	7
TD5. Lack of automatic testing	7
TD1. Product customization	3
TD0. Technical shortcuts	3
TD6. Expensive tests	2
TD3. Lack of multitenancy	2
TD8. Lack of code documentation	2
TD7. Low code coverage in tests	2
TD10. Duplicate code	2
TD4. Hard to maintain simple UX	0

Table 4.2. *Perceived TD types and sum of points*

TD Type	Points
Test TD	11
Requirements TD	10
Code TD	5
Organizational TD	3
Infrastructure TD	2
Documentation TD	2
Design TD	0
Architectural TD	0
Build TD	0
Versioning TD	0
Defect TD	0
Grand Total	30

Table 4.3. *Count of TD motivations presented*

Possible cause of motivations	Count
Budget constraints	5
Time constraints	5
Estimation issues	3
Specification issues	2
Communication issues	1
Design issues	1

4.2 Research Questions

4.2.1 RQ1. What Are the Most Common Types of TD?

The focus group considered the Test TD (11 points) and Requirements TD (10 points) as clearly more significant than other types of TD, as reported in Table 4.2.

4.2.2 RQ2. What Are the Main Causes of the Accumulated TD?

The possible root causes were analyzed with 5 Whys technique [54] by asking why approximately five times for each cause to find the origin of the problem. For each motivation for postponement, we reported possible reasons separated with arrows. Note that a TD item can be postponed for multiple reasons. Possible causes and their counts are summarized in 4.3.

1. Budget constraints (TD1, TD2, TD3, TD7, TD9) and time constraints (TD1, TD4, TD5, TD8, TD9) are the most recurring reasons. Estimation issues (TD1, TD2, TD7) is also a significant cause and closely related to budgeting and timing.
2. Time-related causes (Time constraints), usually related to fast delivery, recurred almost as frequently as budget constraints. It can be speculated that the lack of time depends on the budget.
3. Other causes were not as significant.
4. In some cases, the causes of the TD remain unknown.

- **TD1**: Implementing multiple versions of the same product (Requirements TD, Organizational Debt)
  Causes
  - The customer wanted to use the system for different purposes than originally thought (Specification issues) → Specification issues
  - Prioritizing issues in testing (Prioritizing issues) → Laborious tests → Time constraints → HR constraints → Budget constraints
  - We did not understand the total cost of the change (Budgeting issues) → Budgeting issues → Estimation issues
  - No project management coming from development partners (Budgeting issues) → Budget constraints
  - The bug fixing was much bigger task than thought (Budgeting issues) → Laborious bug fixing → Estimation issues

  - Initial part of project was rushed (Budgeting issues) → Time constraints → Fast delivery

- **TD2**: Disagreement between supplier and team about the Minimum Viable Product specifications and requirements (Requirements TD: Specification issues)
  Causes
  - Budget constraints → Budget constraints
  - Small budget → Budget constraints
  - Big requirement → Estimation issues
  - The UI was designed and supplier did not want to redesign it anymore → Specification issues
- **TD3**: Lack of automatic testing costs more in future (Test TD: Lack of test automation)
  Causes
  - Testing budget was too low to enable creation of automatic testing while developing → Budgeting issues
  - Lack of time to concentrate on delivering fast to client → Fast delivery
- **TD4**: Lack of multitenancy causes budgeting increase and lack of flexibility (Infrastructure TD: Other)
  Causes
  - Not the highest priority → Prioritizing issues → Budget constraints
  - Lack of budget → Budget constraints
- **TD5**: Lack of code documentation (Documentation TD: Lack of code comments)
  Causes
  - Time constraint → Time constraints
  - Developer does not take the time → Developers busy → Prioritizing issues → Time constraints
- **TD6**: Low code coverage in tests causes risks in development and additional work (Test TD: Lack of tests)
  Causes
  - Hard to estimate budget and schedule → Estimation issues
  - Hard to allocate time → Allocation issues → Estimation issues
  - Iceberg issue → Communication issues
  - Developers being testers → HR constraints → Budget constraints
- **TD7**: Technical shortcuts (Code TD: Other)
  Causes
  - Lack of time and money → Budget constraints, Time constraints

  - Not enough dedicated workforce → HR constraints → Budget constraints

- **TD8**: Hard to maintain simple UX (Design TD: Incomplete design specification)
  Causes
  - Hard to find time to redesign UX when functions accumulate → Allocation issues → Time constraints
  - Hard to balance easiness and functionality → Design issues → Complexity
- **TD9**: Duplicated code (Code TD: Duplicated code)
  No causes identified
- **TD10**: Expensive to test (Test TD: Expensive tests)
  Causes
  - No dedicated testers → HR constraints → Budget constraints

4.2.3 RQ3. How to Mitigate TD?

Based on the discussion of the focus group three main aspects that could be improved to mitigate TD were highlighted.

1. *Learning from customers.* First, organizations have to understand what should be built using prototypes and validation with customers.
2. *Careful estimation.* Second, the whole organization should understand the technical boundaries to avoid estimation errors. They should use previous tasks to improve their effort estimation regarding the development of new tasks. Underestimation can cause additional expenses for company. Customers should pay for the overall costs of the system; they tend to pay only for visible costs, which are only the tip of the iceberg. The costs of testing and documentation, which tend to be under the surface, should be made visible to them. The company has to find the right pricing balance in order to remain competitive. Underestimating the amount of work can lead to compromises in less visible costs.
3. *Continuous improvement.* Third, organizations can gradually improve the quality. Deficiencies in development areas should not be postponed. Companies should invest in testing and documentation because their TD's are hindering development and ultimately take up a lot of the developers' precious time. A lack of tests increases the need for manual testing and the risk of regression. Lack of documentation diminishes knowledge and adds tacit knowledge. Evanescence of knowledge will accumulate the costs of testing and documenting over time. Companies have to find the critical point in mitigating TD where benefit is bigger than cost.

5 THREATS TO VALIDITY

In this Section, we will introduce the threats to validity, following the structure suggested by Yin [80] and discussing construct validity, internal validity, external validity, and reliability. Moreover, we will also present the different tactics adopted to mitigate them.

Construct validity is about the correct identification of measures adopted in the measurement procedure (e.g., the questionnaires we used in the studies). Our focus group was based on a comprehensive systematic mapping study of TDM [49].

Internal validity refers to the factors that may have influenced our study. Considering the respondents, interviewed developer had limited time to respond to statements regarding the development. As mentioned in Section 4, CMO and one developer were present only limited time in the focus group. Thus, the personnel are only partly heard and technical issues are only superficially investigated.

External validity are related to the generalization of our findings. Analysis of occurred TD is unreliable because all the aspects cannot be taken into account. Only speculative root causes can be presented and some root causes might not be correct ones. For example, budget constraints can be consequences from estimation errors and specification errors. Estimation errors lead to economical losses. Specification errors lead to rework or useless work. Reasons for specification issues can be communication issues or insufficient knowledge on the subject. In these cases deeper cause is left unsolved. As its weakness, root cause analysis only covers causes that are known. It is expected that the communication in the company is transparent and all the possible causes are mentioned. However, some things are always left unspoken or are forgotten by time. Thus, there can still be causes that remain unknown.

Conclusion validity focuses on how sure we can be that the tasks we adopted are related to the actual outcome we observed [77]. To mitigate this threat, the questionnaires were checked by three experts on empirical studies. Moreover, it was ensured that the subjects of both groups had similar backgrounds and knowledge regarding software development.

6 DISCUSSION

Identifying TD and investigating its possible root causes helped the company to understand their most critical TD items. Conversation helped to determine the causes of accrued TD to enable mitigating TD in the future. Ways to mitigate TD were explored based on the results. Considering all identified TDs, it could be implied that the company has mainly intentional TD. However, intentional TD can be reckless debt taken by developers. Increment of TD is typical especially for SMEs with tight schedule and budget. Nevertheless, based on focus group, the company seems to be transparent and thus, aware of the consequences of its TD. Budget constraints and time constraints (including fast delivery) were considered as most critical root causes of TD.

According to M. Poppendieck et. al. [57], schedule and budget are equal when there are no changes in the personnel. Time constraints can be related to budget constraints when they are caused by HR constraints. However, they do not always relate to budget as more employees do not automatically remove time constraints. According to FP. Brooks [13], work distribution follows Amdahl's law. Thus, the more work is distributable, the more time is saved by adding developers. The required learning curve and the need for more communication lessen the benefit of having more employees. As proven in TD8, level of documentation effects to efficiency of new employees. In addition, lack of tests reduce traceability of requirements and their conformity [57]. Thus, even if the budget is sufficient, time constraints can remain until learning curve is surpassed and the team reaches its optimal performance in group development. As reported by B. Tuckmann [74], in addition to surpassing the learning curve, a new team has to get through forming, norming and storming before getting into performing optimally.

Lowering learning curve, improving group dynamics and improving understanding of the customer can be seen as enablers of continuous improvement. Like in the dynamic model of quality improvement [34], focusing on value demand instead of failure demand decreases the quality costs. In Section 4, we suggested Learning from customer, Careful estimations and Continuous improvement as ways to mitigate TD. In this section we discuss them further.

6.1 Learning from Customer

Learning from the customers is the first answer to RQ3 regarding how to mitigate TD. As stated by one developer, when the organization knows the customer's needs, it helps them to go in the right direction. When there are many customers, User Experience Design becomes more important as a generalized solution has to be created that satisfies everyone's needs at the same time. *"We are kind of having it done by experimenting and communicating more with the customers to understand what they need and we are doing it in an iterative way to solve the customer's problem, but this works until we have only handful of customers."*

Managers can represent VOC to empower the team [57], and increase team efficiency. As stated by a developer, UX designers lacked domain knowledge, and there was a communication gap between managers and designers. Later, CTO helped to remove the gap. "The domain knowledge is the biggest issue. So even when they started with the UX, the domain knowledge that the UX persons had was not that great I guess so that might be the reason. But we have spent their money for UX design only once. The UX was refactored purely by the developers which means that the guys who does not have sales knowledge at all have refactored the UI based on the comments [from the management]. [..] There is a mismatch. The understanding is different. [..] Communicating what they want is not really reaching the developers well. Then having CTO between helps in that way because he understands quite fine so CTO can tell like they get information [from the management] and he translates to software as a UX requirement. Then it's fine. Otherwise there was a big issue until CTO was arranged to this setup."

Idea validation could follow validated learning as suggested by E. Ries [60], prototyping iteratively to get feedback from end-users. According to M. Christel et al. [15], requirements cannot only be gathered, but the customer has to be supported in requirements elicitation iteratively because the customer's understanding is limited. However, the CTO stated that the customer should be consulted only for major decisions and should not be bothered with every minor detail. In TD2, disagreement of MVP requirements, depth-first approach caused waste, because feedback loop was lacking. Instead, iterative breadth-first approach, similar to validated learning, ensures continuous feedback [57].

Prototyping helps to concretize the customer need. According to A. Davis, throwaway prototypes and evolutionary prototypes are commonly used in prototyping [18]. In throw-away prototyping, validation can be as simple as usage of paper prototypes. However, a complex system could be only apprehended with functional high quality prototypes. In web development, separation of concerns in front and back ends allow dividing the development work. Companies can avoid waste with top-down design where back end can be first abstracted with a mock-up while front end can be used to validate the idea. However, it has to be made sure that technical boundaries are understood in the back end to avoid estimation errors. Thus, top-down and bottom-up approaches can be used simultaneously and progressively. In evolutionary prototyping, requirements are implemented

comprehensively and iteratively. Then, there exists a risk that immature prototype ends up in the end product and causes TD for its part. Therefore, quality has to be maintained while validating with users. Development can be incremental while separate features are eventually merged into one. Evolutionary prototyping has been used in the subject company successfully.

Problems caused by a lack of validation were emphasized when the case company outsourced their software development. Sections that are more important for the business than strategically can be outsourced. Outsourcing can allow companies to focus on their core competence, but suppliers have their own interests and all the decisions have to be well-reasoned. The CFO stated about TD2 in Section 4 that the company and the supplier did not agree about the approach of MVP implementation. The company experienced that user tests took two years to implement, which was too long time, and valuable information about user needs was missed during that time. The exact reason of disagreement is not known.

When contracts are well-planned, they ensure mutual benefit in the relationship of vendor and customer. As mentioned in Section 2, a fixed-price contract can lead to vendor delivering less than wanted while optional-scope contract could help to avoid the problem and give an incentive to vendor to learn from customer. [57]

Planning and learning from customer requires good communication. Stakeholders should make sure they consider every aspect of new features, utilizing, for example, ATAM to find possible trade-offs in architecture decisions by formally listening to all stakeholders [36]. A customer approaches the product from a top-down perspective. They cannot see all the technical details related to the implementation of a feature. On the other hand, developers are able to see the bottom-up perspective and know all the technical aspects quite well. However, they might have deficiencies in domain knowledge and cannot value all the customer's needs. Both parties become victims of the Dunning-Kruger effect [41] when they fail to look below the surface. Stakeholders should make sure that they understand each others and all terms discussed have same meanings to everyone.

According to J. Macnamara, 80% of dialogue in organizations is unidirectional speech [50]. Within teams and with stakeholders, monologue fails to take interlocutor in account. It could be said that it loses the opportunity to utilize effectuation, usage of available resources. Instead, it guides to causation, trying to achieve an outcome that might not be optimal for particular environment. Monologue-orientation loses the opportunity to utilize existing information in the organization when the aim to learn from interlocutors can create more optimal results. Giving orders compares to planning ahead with a small amount of information instead of continuously gathering new information and learning from it.

6.2 Careful Estimation

Careful estimation is the second answer to RQ3. SMEs need to use their budget wisely. A limited budget forces a company to generate TD which it hopes to pay back as soon as possible. Payback time can be when the company gets enough funding. The risk of a roadblock through a technical debt bankruptcy increases when new requirements emerge and need attention. These roadblocks lead to the rewriting of existing features, which should be avoided by estimating the costs of TD. Moreover, outsourcing part of the development to consultants, also increase the risk of requirement TD [43] related to misunderstandings [68] and increase the communication overhead [66].

For an SME, budget constraints are inevitable and the company needs ways to cope with its budget. Considering the life-cycle of companies, Nilsson et al. [53] claimed that in the pre-deployment phase, architectural and structural TD should be avoided. Other types of TD such as test and documentation TD can be incurred in that phase. Communication is important especially at the beginning in order to avoid wrong decisions that later become TD.

Any historical analysis of budget constraints is speculative and thus no single reason can be named. Inadequacy in terms of specification validation can drain the budget. TD2, disagreement with the supplier, was considered as one of the most important TD items. The CFO stated that concrete prototypes could have helped validation. Lack of prototyping led to speculation, which wasted time and caused bitterness. Decisions made were not optimal in the long term and caused TD.

Developer claimed that disagreement occurred because of the deficiency in supplier's domain knowledge. According to CFO, the company for their part had a deficiency in technical knowledge. According to developer, repetition of the requirements was not helpful but requirements should have been explained better. Companies outsource on areas where they have lacking knowledge, such as technical areas, but to communicate their perspective to supplier, clients need to ensure that suppliers obtain required domain knowledge; a comprehensive understanding of the customer need. A formal framework such as ATAM [36] can help to prioritize decisions while considering all the perspectives.

Just as with suppliers, companies face challenges in justifying the work to be done with customers. As found out in the results, customers do not see the less visible costs, which should be communicated to them as they improve the long-term health of the system. In addition to their goals, customers have unrecognized needs that the product has to fulfill. Quality increases customer satisfactory and thus, increases customer loyalty. Therefore, customer has to have motivation to pay for a product that is not only usable and effective in completing the goals but is also efficient and provides satisfying UX. The CFO gave the following example: *"We had an issue that one of our customers wanted to modify the questions. [...] It was quite a big change and they said that 'no, we won't pay that much' and then we said we cannot do it. They were not very happy but we had no choice. It*

was too expensive and the client did not see any value in that."

Careful estimation avoids risks. In addition to communication issues, estimation errors can be reasons that drain the budget. As mentioned in Section 4, underestimation is unprofitable for the company. Estimation errors occur because of unpredictable complexity of a task. Developers might not see all the sub-tasks concerning a new task when doing the estimation at the beginning of the development of a new task. Every new task is unique and has little in common with the previous tasks. A little knowledge of the subject makes developers overconfident, which leads to them underestimating the amount of work. Again, only the tip of the iceberg is seen and a new task is seen as simpler than it actually is in the end.

Dunning-Kruger effect [41], overconfidence of own competence, can explain occurrence of inadvertent and reckless debt. Developers might give too optimistic estimations, and when they are found not to be met, taking TD becomes relevant. Furthermore, developers might fail to predict overall consequences of their decisions. TD and the effort it causes will be underestimated. Awareness of cognitive biases helps to mitigate TD.

When pricing and schedule are unrealistic, development will focus only on the most critical areas. Pessimism in estimation could help to improve quality, but the challenge is to maintain competitiveness with bigger companies that have economies of scale and can use their capital to fund all the various aspects. Companies can find estimation challenging, as stated by the CFO about TD1, where the total cost was approximately 10 times bigger than the estimated cost.

Companies can improve in finding critical point in estimates by time meanwhile preparing for estimation errors. Float (or slack) in task estimates help to form realistic critical path for the work. As in Parkinson's law [55], task duration might be overestimated since after some point, finishing a task requires the same time regardless of the allocated time. Meanwhile, as in Hofstadter's law [32], task duration might be underestimated since a task requires more time than estimated although the estimator would be aware of estimation challenges.

6.3 Continuous Improvement

Continuous improvement is the third answer to RQ3. As continuous improvement is a dynamic cost model, improvement activities provide learning opportunity which lowers quality costs [34], and helps developers to improve their design approaches [27]. It is wise for companies to find time for TD repayment, because TD is a waste, as it hinders development, and causes additional work [65] and extraneous cognitive load [62]. Ignoring TD increases incremental debt, and presence of TD decreases productivity through decay of maintainability and extensibility [73]. Response time to customers increases as failure demand is taken away from value demand [59]. Furthermore, product quality affects to brand image [73]. Prioritization of TD repayment could follow cost-benefit

analysis, portfolio approach and TD value estimates [4].

Focus group considered Test TD as most important TD type closely followed by Requirements TD. According to a developer, the quality of the code has suffered from a lack of unit tests. Testing is needed to validate conformance to requirements, which for its part also prevents Requirements TD. Tests provide scaffolding for development and communicate intention [57]. Knowledge loss in personnel changes and as time passes causes relearning [58] [62], which can be avoided with proper tests and documentation. Existence of tests confirms that development is going to right direction. One developer stated: *"Requirements were not written anywhere and if you touch and you happen to break something it's even hard to regulate what's broken until it gets into the customer's hands."*

Lack of tests create new risks which could be avoided by gradual implementation of tests. However, SMEs can find difficult to afford tests although their benefits would exceed costs in long term, as focus group stated regarding Test TD5, TD6 and TD7 in the results. The company outsourced software development in the beginning and the supplier had personnel with certain roles such as developer, tester and designer. As mentioned in TD5, the company experienced that testers did not understand the context of use, and lacked domain knowledge. It can be said that cooperative, cross-functional team might obtain required domain knowledge easier, because they work in a wider area.

According to E. Dijkstra [22], an efficient developer should not introduce bugs in the first place but write provable code that complies with the requirements. TDD as an extreme approach requires simultaneous development of features and their tests. Benefit is that features will comply to requirements from the start. However, most important is to iteratively validate new features and testing can be done only when conformity to the customer need is confirmed. Nevertheless, testing and development, in the same manner as design and implementation, should not be separated but should actively share information with each other [22]. Cooperation also decreases time wasted on queuing [57].

As changes in requirements are inevitable, software needs to be designed for change [59] with modularization, separation of concerns and information hiding as stated by D. Parnas [56].

Companies need to find the golden mean in quality improvement. Testing can be seen as a way to maintain software quality. Testing, documenting, and bug fixing are ways to reduce waste in software development. As stated by CFO about TD5 in the results, testing and fixing bugs become more difficult over time when software entropy increases.

Test automation might not be affordable for SMEs because of high upfront costs. However, as in TD7, low code coverage in tests caused additional work and thus additional costs. SMEs can consider different factors affecting to profitability of test automation. According to a systematic mapping study by V. Garousi et. al., test automation is profitable most importantly when frequent regression testing is needed (mentioned in 44 studies), tests bring economic benefits (43 studies), and interfaces to tests are unlikely to change

(39 studies). Test automation might not be profitable when the System Under Test (SUT) faces major modifications in the future (39 studies). Neither when tests have risk of computer errors (28 studies) or require a lot of maintenance (18 studies). Nor when organization has tight schedule, budget pressure or large change resistance (11 studies), or when tests require human judgment or have fragile comparison (10 studies). Last, when SUT is tightly integrated or complex (6 studies). [28]

Risk management can help in considering importance of test automation. If lack of tests risks usability, reliability or business, then systematic testing is considerable. In long term, test automation saves time compared to manual testing. However, based on findings by V. Garousi et. al. [28], amount of work required by implementation of test automation depends on development process. In agile process, changes are frequent which destabilizes the system. Companies can aim to stabilize the system by maintaining tests when changes occur. Nevertheless, based on TD5, TD6 and TD7, SMEs find difficult to find time and budget for test automation if customers don't see the benefit. Cost-benefit analysis can help to communicate long term benefit to customers.

When TD occurs, its payback is related to quality improvement; the critical point is found through cost-benefit analysis. Quality can be improved gradually. Cost-benefit analysis can help to improve Pareto efficiency in similar ways as in [76]. Finding the golden mean where the cost-benefit ratio is the lowest can be the most beneficial goal.

7 CONCLUSION

In this work, we performed an empirical study to understand the main reasons for Technical Debt in a SME company, the problems it created, and how we mitigated it. The goal was to understand what happened in the past, so as to avoid making the same mistakes again, or to make reasoned choices. As seen in table 4.1, our participants considered the most significant TD items to be disagreement with supplier (7 points) and lack of creation of test automation (7 points). The most significant types of TD were Test and Requirements. Possible root causes named for these TD items were budget constraints, estimation issues, specification issues, and fast delivery. Overall, as seen in table 4.3, the most important root causes were considered to be budget constraints (5 occurrences), time constraints (5 occurrences), and estimation issues (3 occurrences).

Finding consensus requires building bridges within teams and between stakeholders. The goal is to understand the reasons behind occurred TD and ways to mitigate it. Motivation to mitigate TD comes from the reason that the consequences of TD is emphasized in SMEs which have limited budget. SMEs are also more liable to take TD than bigger companies because limited budget forces them to make more sacrifices. However, TD is only harmful when it's left unpaid as stated by W. Cunningham [17].

Attempting to build a connection to management theory helps to understand the issue of TD in depth. Based on the analysis of the results and related work, the following methods can be used to mitigate TD:

- *Learning from customers* - prototyping with the customers to find the right direction and communicating efficiently with the stakeholders;
- *Careful estimation* - improving meta-cognition to learn estimation;
- *Continuous improvement* - using limited budget and time wisely to bring value.

Another interesting result from our study occurred in TD2, disagreement with the supplier. Requirements were not validated properly with the company nor their customers at the beginning when the product was outsourced to an external supplier. Communication issues and contract form were most potential reasons we found. Both parties need to be motivated to understand each others' perspectives. Using an optional-scope contract encourages to seek for common good and in-depth understanding of customer need, learning from customers. [57].

Furthermore, during outsourcing the company experienced that the supplier's personnel

lacked domain knowledge. The personnel had certain roles as developers, testers and designers, and did not have wide enough knowledge of the context of use. Cooperative, self-directed cross-functional team could perform better, since it can obtain domain knowledge easier, performing in different roles. Empowering the team and giving the personnel more responsibility provides the whole organization required understanding for the current situation [57] and proactivity for the future, creating an ambidextrous organization [75].

Moreover, estimation issues hindered future development in TD1, product customization. Developers underestimated the time needed for testing and bug fixing. As estimation errors are harmful to budgeting and scheduling, careful estimation, awareness of one's own competence and transparency in communication can avoid risks in the future.

Ultimately, continuous improvement and learning is beneficial in long term. The company found several areas for improvement without finding resources for them. SMEs have tight budget and schedule whereupon TD prioritization is particularly important. Related work provides several ways, using e.g. cost-benefit analysis or portfolio approach. Opportunities and risks has to be seen. However, ROI or costs of TD can be difficult to define and thus, risk management is required for prioritization. Nevertheless, taking prudent TD has enabled progression for the company's business.

This work provides an overview of the main issues related to Technical Debt in our company. However, we are aware of different threats that may have influenced the results. Some participants might not have reported some Technical Debt issues for different reasons. The presence of the company's technical management (CTO, CFO, and CMO) could have influenced the answers of the developers. The focus group was conducted over a period of two hours, and therefore we probably have not reported all the issues that occurred during the history of the company, but only the most recent or the most significant ones.

Further studies are needed to create a stronger bond between the effects of validation and estimation on the one hand and budgeting and scheduling on the other hand. Benchmarks of our estimations with existing dataset [45] and adopting TD management tools widely used by competitors [47][46] could be a viable solutions to mitigate this threat. Understanding these laws also requires interdisciplinary studies that combine computing, quality management, and psychology. A continuous quality management approach [35][48] could help to prevent TD. Moreover, management studies help to develop better processes, while psychology and organizational studies can explain why estimation errors occur. Understanding root causes by looking at them through these fields will result in better knowledge of TD and help SMEs avoid pitfalls, thereby enabling them to be even more successful.

The results of this work have been published to the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM) [44].

REFERENCES

[1] Y. Akao. QFD: Past, present, and future. *International Symposium on QFD*. Vol. 97. 2. 1997, 1–12.

[2] E. Allman. Managing technical debt. *Commun. ACM* 55.5 (2012), 50–55.

[3] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou, P. Avgeriou, P. Abrahamsson, A. Martini, U. Zdun and K. Systa. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. *MTD '16*. Oct. 2016, 9–16.

[4] A. Ampatzoglou, A. Ampatzoglou, A. Chatzigeorgiou and P. Avgeriou. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology* 64 (2015), 52–73.

[5] R. D. Austin. *Measuring and managing performance in organizations*. Addison-Wesley, 2013.

[6] S. E. Bailey, S. S. Godbole, C. D. Knutson and J. L. Krein. A Decade of Conway's Law: A Literature Review from 2003-2012. *2013 3rd International Workshop on Replication in Empirical Software Engineering Research*. IEEE. 2013, 1–14.

[7] V. R. Basili, G. Caldiera and H. D. Rombach. The Goal Question Metric Approach. *Encyclopedia of Software Engineering* (1994).

[8] K. Beck and D. Cleal. Optional Scope Contracts. *White Paper, Three Rivers Institute* (1999).

[9] K. Beck, M. Beedle, A. Van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries et al. Manifesto for agile software development. (2001).

[10] T. Besker, A. Martini, R. E. Lokuge, K. Blincoe and J. Bosch. Embracing Technical Debt, from a Startup Company Perspective. *ICSME 2018*. 2018.

[11] S. Blank and B. Dorf. *The startup owner's manual: The step-by-step guide for building a great company*. BookBaby, 2012.

[12] S. Blank and P. Newell. What your innovation process should look like. *Harvard Business Review* (2017).

[13] F. P. Brooks Jr. The mythical man-month (anniversary ed.) (1995).

[14] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya et al. Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM. 2010, 47–52.

[15] M. G. Christel and K. C. Kang. *Issues in requirements elicitation*. Tech. rep. Carnegie-Mellon Software Engineering Inst, 1992.

[16] A. Clay Shafer. Infrastructure Debt: Revisiting the Foundation. *Cutter IT Journal* 23.10 (2010), 36.

[17] W. Cunningham. The WyCash portfolio management system. *SIGPLAN OOPS Messenger* 4.2 (1993), 29–30.

[18] A. M. Davis. Operational prototyping: A new development approach. *IEEE software* 9.5 (1992), 70–78.

[19] S. De Toledo, A. Martini, A. Przybyszewska and D. Sjoberg. Architectural Technical Debt in Microservices. A case study in a large company. *TechDebt 2019*. 2019.

[20] W. E. Deming. *Out of the Crisis*. MIT press, 2018.

[21] W. E. Deming. *The new economics for industry, government, education*. MIT press, 2018.

[22] E. W. Dijkstra. The humble programmer. *Commun. ACM* 15.10 (1972), 859–866.

[23] P. Domingos. The role of Occam's razor in knowledge discovery. *Data mining and knowledge discovery* 3.4 (1999), 409–425.

[24] N. A. Ernst. On the role of requirements in understanding and managing technical debt. *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press. 2012, 61–64.

[25] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord and I. Gorton. Measure it? manage it? ignore it? software practitioners and technical debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, 50–60.

[26] M. Fowler. *StranglerApplication*. 2004. URL: `https://www.martinfowler.com/bliki/StranglerApplication.html`.

[27] M. Fowler. Technical debt quadrant. *Martin Fowler* (2009), 14–.

[28] V. Garousi and M. V. Mäntylä. When and what to automate in software testing? A multi-vocal literature review. *Information and Software Technology* 76 (2016), 92–117.

[29] Y. Guo and C. Seaman. A portfolio approach to technical debt management. *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. 2011, 31–34.

[30] S. Hajikazemi, A. Ekambaram, B. Andersen and Y. J. Zidane. The Black Swan–Knowing the unknown in projects. *Procedia-Social and Behavioral Sciences* 226 (2016), 184–192.

[31] F. Herzberg et al. *One more time: How do you motivate employees*. 1968.

[32] D. R. Hofstadter et al. *Gödel, Escher, Bach: an eternal golden braid*. Vol. 20. Basic books New York, 1979.

[33] A. Hunt and D. Thomas. The Art in Computer Programming. *The Pragmatic Programmers, LLC* (2001).

[34] C. D. Ittner. Exploratory evidence on the behavior of quality costs. *Operations Research* 44.1 (1996), 114–130.

[35] A. Janes, V. Lenarduzzi and A. C. Stan. A Continuous Software Quality Monitoring Approach for Small and Medium Enterprises. *International Conference on Performance Engineering Companion*. 2017.

[36] R. Kazman, M. Klein, M. Barbacci, T. Longstaff, H. Lipson and J. Carriere. The architecture tradeoff analysis method. *Inter. Conference on Eng. of Complex Computer Systems*. 1998, 68–78.

[37] T. Klinger, P. Tarr, P. Wagstrom and C. Williams. An Enterprise Perspective on Technical Debt. *MTD '11*. 2011, 35–38.

[38] E. Klotins, M. Unterkalmsteiner, P. Chatzipetrou, T. Gorschek, R. Prikladnicki, N. Tripathi and L. Pompermaier. Exploration of technical debt in start-ups. *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*. IEEE. 2018, 75–84.

[39] J. Kontio, J. Bragge and L. Lehtola. The Focus Group Method as an Empirical Tool in Software Engineering. *Guide to Advanced Empirical Software Engineering*. Ed. by F. Shull, J. Singer and D. I. K. Sjøberg. 2008, 93–116.

[40] P. Kruchten, R. L. Nord and I. Ozkaya. Technical debt: From metaphor to theory and practice. *Ieee software* 29.6 (2012), 18–21.

[41] J. Kruger and D. Dunning. Unskilled and unaware of it: how difficulties in recognizing one's own incompetence lead to inflated self-assessments. *Journal of personality and social psychology* 77.6 (1999), 1121.

[42] V. Lenarduzzi and D. Taibi. MVP Explained: A Systematic Mapping Study on the Definitions of Minimal Viable Product. *42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2016, 112–119.

[43] V. Lenarduzzi and D. Fucci. Towards a Holistic Definition of Requirements Debt. *13th International Symposium on Empirical Software Engineering and Measurement*. Sept. 2019.

[44] V. Lenarduzzi, T. Orava, N. Saarimäki, K. Systä and D. Taibi. An Empirical Study on Technical Debt in a Finnish SME. *13th International Symposium on Empirical Software Engineering and Measurement*. Sept. 2019.

[45] V. Lenarduzzi, N. Saarimäki and D. Taibi. The Technical Debt Dataset. *Int. Conf. on Predictive Models and Data Analytics in software engineering (PROMISE'19)*. Sept. 2019.

[46] V. Lenarduzzi., A. Sillitti and D. Taibi. Analyzing Forty Years of Software Maintenance Models. *International Conference on Software Engineering Companion (ICSE-C)*. 2017.

[47] V. Lenarduzzi, A. Sillitti and D. Taibi. A Survey on Code Analysis Tools for Software Maintenance Prediction. *Software Engineering for Defence Applications - SEDA*. 2019.

[48] V. Lenarduzzi, C. Stan, D. Taibi, D. Tosi and G. Venters. A Dynamical Quality Model to Continuously Monitor Software Maintenance. *11th European Conference on Information Systems Management*. 2017.

[49] Z. Li, P. Avgeriou and P. Liang. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), 193–220.

[50] J. Macnamara. Creating an "architecture of listening" in organizations. *The basis of engagement, trust, healthy democracy, social equity, and business sustainability. University of Technology Sydney, Sydney* (2015).

[51] A. Martini, J. Bosch and M. Chaudron. Investigating Architectural Technical Debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology* (2015), 237–253.

[52] S. McConnell. Managing technical debt. *Construx Software Builders, Inc* (2008), 1–14.

[53] H. Nilsson and L. Petersson. *How to Manage Technical Debt in a Lean Startup*. 2013.

[54] T. Ohno. *Toyota production system: beyond large-scale production*. crc Press, 1988.

[55] C. N. Parkinson and R. C. Osborn. *Parkinson's law, and other studies in administration*. Vol. 24. Houghton Mifflin Boston, 1957.

[56] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM* 15.12 (1972), 1053–1058.

[57] M. Poppendieck and T. Poppendieck. *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003.

[58] M. Poppendieck and T. Poppendieck. *Implementing lean software development: From concept to cash*. Pearson Education, 2007.

[59] M. Poppendieck and T. Poppendieck. *Leading lean software development: Results are not the point*. Pearson Education, 2009.

[60] E. Ries. *The lean startup.* Crown Books, 2011.

[61] S. D. Sarasvathy. Causation and effectuation: Toward a theoretical shift from economic inevitability to entrepreneurial contingency. *Academy of management Review* 26.2 (2001), 243–263.

[62] T. Sedano, P. Ralph and C. Péraire. Software development waste. *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, 130–140.

[63] P. M. Senge. *The fifth discipline fieldbook: Strategies and tools for building a learning organization*. Crown Business, 2014.

[64] D. I. Sjøberg, A. Johnsen and J. Solberg. Quantifying the effect of using kanban versus scrum: A case study. *IEEE software* 29.5 (2012), 47–53.

[65] M. G. Stochel, M. R. Wawrowski and M. Rabiej. Value-based technical debt model and its application. *ICSEA 2012, The Seventh International Conference on Software Engineering Advances*. 2012.

[66] D. Taibi, V. Lenarduzzi, M. O. Ahmad and K. Liukkunen. Comparing Communication Effort Within the Scrum, Scrum with Kanban, XP, and Banana Development Processes. *21st International Conference on Evaluation and Assessment in Software Engineering*. EASE'17. 2017.

[67] D. Taibi, V. Lenarduzzi, M. O. Ahmad and K. Liukkunen. Comparing communication effort within the scrum, scrum with kanban, xp, and banana development processes. *Proceedings of the 21st International Conference on Evaluation and Assessment in Software Engineering*. ACM. 2017, 258–263.

[68] D. Taibi, V. Lenarduzzi, A. Janes, K. Liukkunen and M. O. Ahmad. Comparing Requirements Decomposition Within the Scrum, Scrum with Kanban, XP, and Banana

Development Processes. *Agile Processes in Software Engineering and Extreme Programming*. 2017.

[69] N. N. Taleb. *The black swan: The impact of the highly improbable*. Vol. 2. Random house, 2007.

[70] K. Tate. *Sustainable software development: an agile perspective*. Addison-Wesley Professional, 2005.

[71] K. W. Thomas and B. A. Velthouse. Cognitive elements of empowerment: An "interpretive" model of intrinsic task motivation. *Academy of management review* 15.4 (1990), 666–681.

[72] F. Thompson. Public economics and public administration. *PUBLIC ADMINISTRATION AND PUBLIC POLICY* 65 (1998), 995–1064.

[73] E. Tom, A. Aurum and R. Vidgen. An exploration of technical debt. *Journal of Systems and Software* 86.6 (2013), 1498–1516.

[74] B. W. Tuckman. Developmental sequence in small groups. *Psychological bulletin* 63.6 (1965), 384.

[75] M. L. Tushman and C. A. O'Reilly III. Ambidextrous organizations: Managing evolutionary and revolutionary change. *California management review* 38.4 (1996), 8–29.

[76] S. H. Vathsavayi and K. Systä. Technical Debt Management with Genetic Algorithms. *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. Aug. 2016, 50–53. DOI: 10.1109/SEAA.2016.43.

[77] C. Wohlin, P. Runeson, M. Höst, M. Ohlsson and. A. W. B. Regnell. *Experimentation in Software Engineering: An Introduction*. 2000.

[78] D. C. Wood. *The executive guide to understanding and implementing quality cost programs: reduce operating expenses and increase revenue*. ASQ Quality Press, 2007.

[79] A. D. Yakup and Z. Sevil. A theoretical approach to the concept of the costs of quality. *International Journal of Business and Social Science* 3.11 (2012).

[80] R. Yin. *Case Study Research: Design and Methods, 4th Edition (Applied Social Research Methods, Vol. 5)*. 4th. SAGE Publications, Inc, 2009.

[81] W. I. Zangwill and P. B. Kantor. Toward a theory of continuous improvement and the learning curve. *Management Science* 44.7 (1998), 910–920.