DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Identifying and Assessing the Effects of Domain Level Technical Debts in Practice

Joonas Palm

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Identifying and Assessing the Effects of Domain Level Technical Debts in Practice

Identifizierung und Bewertung der Auswirkungen von technischen Schulden auf Domänenebene in der Praxis

Author:	Joonas Palm
Supervisor:	Prof. Dr. Florian Matthes
Advisor:	Ömer Uludağ, M. Sc.
Submission Date:	15 April 2021

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 06 April 2021 Joonas Palm

Acknowledgments

At this point I would like to thank my supervisor Prof. Dr. Florian Matthes, my advisor Ömer Uludağ, and my two mentors Dr. Marcus Ciolkowski and Prof. Dr. Harald Störrle from QAware GmbH. We had a very close collaboration with Marcus and Harald at the time of writing this master's thesis. They gave me very valuable feedback and I had the opportunity to learn a lot from them throughout this process. Thanks to the efforts of Marcus and Harald, we were able to collect the data that was analyzed in this thesis. I am especially thankful that they agreed to supervise me, regardless of the huge load of work they already had. Thanks again for the cooperation!

I would also like to thank all the great friends I met during my studies at the Technical University of Munich. These studies were fun and adventurous with you. I would like to thank my friends and relatives in Estonia, who, despite the long distance, were constantly in contact with me. True friendship is not harmed by long distances!

Last but not least, I would also like to thank my parents Kärt Palm and Tõnu Palm and my dear little sister Lee Palm for always being there for me and supporting me during my studies.

Abstract

Technical Debt (TD) has been accepted as a useful metaphor in the world of software engineering. The concept includes software problems that are comparable in course and nature to financial debt. TD is not always directly caused by code-related artifacts, but can often be caused by something more abstract. The recently defined term "Domain Level TD" is a new type of TD that takes a domain-oriented perspective on a software system. It describes TD that arises when there is a quality gap between the software system and its domain.

The objective of this thesis is to describe instances of domain level TD from practice as case studies and thereby map its characteristics and effects. Additionally, the goal is to develop a general procedure for identifying domain level TD and to compare different levels of TD in a software system to each-other on the basis of economic and technological impact.

The study indicates that domain level TD is often relatively expensive. Compared to implementation level TD, such as code TD or design TD, domain level TD cannot be identified by simply analyzing the source code of the system. Domain knowledge and expertise are essential for being able to identify and manage domain level TD. It does not always primarily affect the developers of the system, but often also other stakeholders, like for example the users and the business departments. A checklist of 20 questions has been proposed as a way for identifying domain level TD in a system. A list of TD items, which was collected from a project from the industry, has been made public for any future research.

Kurzfassung

Technische Schulden wurde als nützliche Metapher in der Welt der Softwareentwicklung akzeptiert. Das Konzept umfasst Softwareprobleme, die im Verlauf und Art mit Finanzschulden vergleichbar sind. Technische Schulden werden nicht immer direkt durch Code-bezogene Artefakte verursacht, sondern können häufig durch etwas Abstrakteres verursacht werden. Der kürzlich definierte Begriff "Fachschulden" ist ein neuer Typ von technischen Schulden, der eine domänenorientierte Perspektive auf ein Softwaresystem einnimmt. Es beschreibt technische Schulden, die entstehen, wenn zwischen dem Softwaresystem und seiner Domäne eine Qualitätslücke besteht.

Ziel dieser Arbeit ist es, Instanzen von Fachschulden aus der Praxis als Fallstudien zu beschreiben und dabei deren Eigenschaften und Auswirkungen abzubilden. Darüber hinaus besteht das Ziel darin, ein allgemeines Verfahren zur Identifizierung von Fachschulden zu entwickeln und verschiedene Ebenen von technischen Schulden in einem Softwaresystem auf der Grundlage der wirtschaftlichen und technologischen Auswirkungen miteinander zu vergleichen.

Die Studie zeigt, dass Fachschulden häufig relativ teuer sind. Im Vergleich zu technischen Schulden auf Implementierungsebene, wie z. B. Codeschulden oder Designschulden, können Fachschulden nicht identifiziert werden, indem einfach der Quellcode des Systems analysiert wird. Fachwissen und -kompetenz sind unerlässlich, um Fachschulden identifizieren und verwalten zu können. Dies betrifft nicht immer in erster Linie die Entwickler des Systems, sondern häufig auch andere Interessengruppen, wie beispielsweise die Benutzer und die Geschäftsabteilungen. Eine Checkliste mit 20 Fragen wurde vorgeschlagen, um Fachschulden in einem System identifizieren zu können. Eine Liste von techischen Schulden, gesammelt aus einem Projekt aus der Industrie, wurde für zukünftige Forschungsarbeiten veröffentlicht.

Contents

1. Introduction

It was the spring of 1992, when Ward Cunningham published an experience report that enriched the Computer Science community with yet another substantial keyword - Technical Debt (TD) [1]. He described it as: "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. . . . The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt" [1]. With these sentences Cunningham deftly creates a vehicle for communicating about software development issues with non-technical stakeholders, especially managers, who are often more familiar with the financial world [2, 3, 4]. Also, Cunningham clearly states that TD should not be demonized, as it can be beneficial in certain situations, e.g. reducing time to market. In this sense it can be compared to financial debt, where it is often necessary for larger investments, like purchasing real estate. TD is often the result of compromises made between the technical quality of software and business goals [2, 5, 6]. However, Cunningham also warns that the fruits of TD should be enjoyed with caution, as large amounts of it can have serious consequences and even bring the whole organization to a halt [1].

The negative effects of TD and a clear reason why it should be further researched, can be found in the CISQ report on the cost of poor quality software in the US [7]. They state that approximately 2.84 trillion US dollars were spent due to software quality issues in 2018 [7]. Over a quarter of this money (37,46%) was given out on losses from software failures, 21,42% on legacy system problems, and 18,22% directly on TD [7]. This overall sum corresponds to the GDP of the United Kingdom in the same year [8]. Due to these severe symptoms and pains the TD metaphor has been further researched and investigated by the Computer Science community since its inception by Cunningham. This growing trend is well visualized by the amount of publications that contain the word "Technical Debt" in its abstract or title. Since 2008, this figure has started to rise significantly and has reached close to 250 by 2020 (see Figure 1.1).

Researchers have been working on defining clear boundaries for the definition of TD and also on the identification, assessment, and management of it. A recent commonly accepted definition for TD has been proposed by Brown et al.: "the gap between the current state of a software system and some hypothesized "ideal" state in which the system is optimally successful in a particular environment" [9]. This definition is relatively vague and implies that TD can be found in all corners of the software system, ranging from code artifacts and low-level design patterns to high-level architecture,

documentation, and development processes. However, most of the TD related research concentrates on design issues found on the lower implementation level [2, 10, 11, 12]. A possible reason for this might be that code deficiencies are more visible and can be identified with the help of automated analysis tools [10, 12]. At the same time, identifying high-level issues is often more complex, as it requires thorough manual reviews and a wider overview of the system and its context [13]. Yet low-level implementation issues often have a much smaller negative impact and are easier to fix than high-level TD [2]. It is easier to break down a very long method or remove duplicates, than to modify the architecture of a software system.

This master's thesis aims at reducing this deficit by identifying and assessing the effects of domain level TD in practice. The term "Domain Level TD" was coined in 2019, as two researchers noticed that there exists yet another type of TD that has not been discussed by literature so far [2]. They define it as "the misrepresentation of the application domain by an actual system" [2]. Specifically, domain level TD focuses on the question if the system does what it should optimally rather than how the internals have been implemented. Until now, to the best of our knowledge, there has not been any further research on this part of TD and our goal is to fill this gap by conducting empirical case studies that describe examples of domain level TD in practice. The aim is to elicit common characteristics and effects of domain level TD from these case studies. This will hopefully support the identification and measurement of it in future software projects. In addition to that, this thesis makes use of TD data, collected at a software company, to describe the proportions and impacts of different levels of TD.

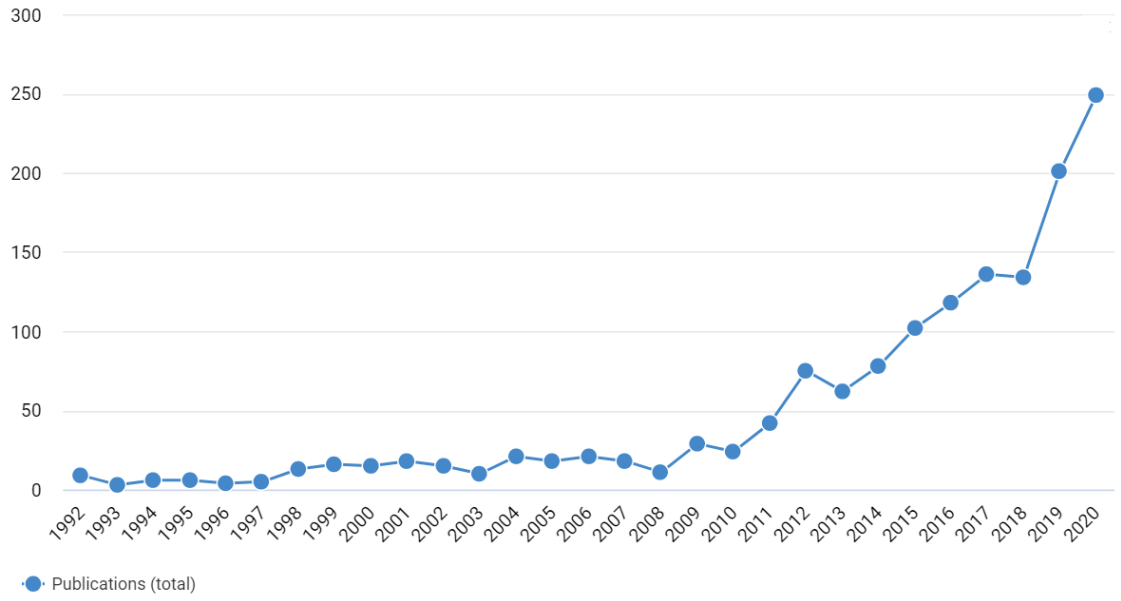

Figure 1.1.: The number of publications containing "Technical Debt" in the title or in the abstract [14].

1.1. Problem Statement

In 2019, in an article published at the annual TechDebt conference [15], Harald Störrle and Marcus Ciolkowski identified and described a gap in TD research [2]. The gap, labelled "Domain Level TD" or simply "Domain Debt", consists of a new type of TD, which, to the best of our knowledge, has not been researched sufficiently to date. Most of the academic TD research has been focusing on TD at the level of technology and less at other abstraction levels of the software system [2, 10, 11, 12]. In their article, Störrle and Ciolkowski provide an initial definition for the term "Domain Level TD" and describe three examples of it [2]. This is only a beginning, as the initial definition is relatively vague and does not clearly list specific characteristics and effects of domain debt, which could potentially even lead to a way of identifying and measuring it. Therefore, more research is needed to continue mapping the phenomenon.

The goal of this thesis is to analyze a list of TD items collected at QAware GmbH, a software company in Munich, by evaluating their economic and technological impacts and interpreting the findings. In addition to that, the objective is to elicit instances of domain level TD from the list and other projects and to further describe and investigate their characteristics and effects. The characteristics and effects possibly allow us to put together a general checklist for identifying items of domain level TD. This preliminary checklist is then consolidated and elaborated with the help of feedback and reviews from software experts.

In this master's thesis we try to answer the following Research Questions:

RQ1 What are the characteristics of domain level TD in practice? This question tries to map and further define the term domain level TD, by observing its characteristics and effects in practice. The characteristics are elicited from empirical case studies that describe instances of domain level TD. These instances are in turn selected from different projects at QAware GmbH. The goal is to analyze the commonalities and differences between these different examples of domain level TD, to better understand its causes, effects, and indicators. In addition to that, the objective is to examine, if the characteristics and definitions proposed in [2] are identifiable and valid in practice.

RQ2 Can domain level TD be measured, and if so, how? The goal of this question is to determine, if there exists a general way of measuring domain level TD in a system. By measuring we mean the detection of domain level TD. This is done by compiling a checklist or a collection of questions that forms the basis for a procedure that can be used to determine the degree to which a certain instance of TD can be considered as domain debt. For instance, the amount of TD at implementation level can often be automatically measured by various static code analysis tools that directly translate the detected issues into an amount of time that a developer would need to resolve them. However, it is unclear if this can also be done for other more abstract types of TD, including domain level TD. A question-based procedure for eliciting domain debt would be a first step towards automation. This question is directly based on RQ1, which

inspects the characteristics of domain debt. We try to use the identified characteristics as a basis for questions in the procedure for identifying domain debt.

RQ3 What are the proportions of different levels of TD in terms of number and size of issues, economic, and technological impact? This question does not only focus on domain level TD, but also considers other types of TD. The list of TD, forming the basis of this thesis, collected from a real-world software project, provides a unique possibility for being able to assess the proportions and impacts of TD in practice. We try to collect and analyze information on costs and consequences of TD items and illustrate how different types of TD affect the project and the software system. This could possibly provide some empirical evidence of what the most common levels of TD are in a system and how they affect the stakeholders. In addition, it makes it possible to compare domain level TD to other levels of TD.

1.2. Outline

The chapter 2, "Related Work", describes the search strategy used for collecting literature and studies reporting on TD. In addition to that, it provides an overview of the previous literature on TD and more specifically on the different types of TD, on domain level TD, and on the identification and management of TD.

The chapter 3, "Research Methodology", describes the data collection and research methodology used in this thesis. It provides an overview of the overall research plan, how data was collected, and how it was analyzed to answer the Research Questions.

The chapter 4, "Results", provides an overview of the results of the data collection phase. It presents the results from the domain level TD case studies and from the list of TD items, which was collected from a project from the industry.

The chapter 5, "Discussion of Results", analyzes and discusses the results of the studies and answers the Research Questions. It discusses the characteristics and effects of domain level TD in practice, proposes a checklist for identifying domain level TD, and discusses the proportions and impacts of different levels of TD in a project from the industry. Finally, the limitations of the study are listed.

Eventually, chapter 6 and chapter 7 conclude the results and give an outlook on future work.

2. Related Work

This chapter provides an overview of the previous literature on TD and more specifically on the different types of TD, on domain level TD, and on the identification and management of TD.

2.1. Search Strategy

An electronic search was conducted to find studies reporting on different types of TD and TD identification and management approaches. The search strategy involved the use of search phrases containing the words "technical debt", "technical debt types", "domain level technical debt", "technical debt literature review", "technical debt identification", "technical debt management", and "TechDebt conference". The international TechDebt conference is held annually and provides the opportunity for researchers and practicioners to come together and discuss recent advancements and issues in the field of TD [15]. It is sponsored by the major software engineering communities ACM SIGSOFT and IEEE TCSE [15]. DBIS (TUM Datenbank-Infosystem), TUM OPAC (`https://www-ub-tum-de.eaccess.ub.tum.de/tum-opac`), IEEE Xplore (`https://ieeexplore.ieee.org`), and Google Scholar (`https://scholar.google.com`) were systematically searched for studies published between inception and December 2020.

2.2. The Technical Debt Metaphor

In a recent study in 2018 Besker et al. found out that developers on average waste nearly a quarter of their development time on something called TD [11]. This time is mostly spent on additional testing, analysis, and refactoring [11]. And even more bad news: TD tends to spread like a viral infection if not handled properly, because in almost a quarter of cases when developers confront TD, they tend to grow and spread it even further [11]. So what is this TD? Is it as utterly evil as it seems?

The metaphor of TD is partially based on the first and second law of Lehman, where he states that a software system must continually change in order to meet the volatile requirements and as a result the complexity of the evolving system also increases [16, 5]. If the system does not keep up with the requirements, it incrues TD in the form

of not fulfilling the changing needs of its users or not employing the most up-to-date technologies. However, if the system does change, it usually takes on TD in the form of quality and design issues, caused by the increasing complexity of it. If one may say so, TD is written into the scientific laws of Software Engineering. This means that TD cannot always be avoided and is a natural part of software development [17, 18, 19].

It is often unreasonable and also impossible to try to satisfy one's perfectionist needs and develop the "ideal" software system. As Klaus Schmid mentions in his article on the limitations of the TD metaphor: "There is nothing like a technical-debt-free system" [20]. It is often also pointless to strive towards this "debt-free" system, because as we will explain later, TD can also be a good investment. However, it is vital to know about the existence of TD in a system, where it hides, and what its causes and effects are. The fact that TD is sometimes inevitable does not mean that there is no point in planning and designing ahead of development, as some types of TD can be so costly that they should always be bypassed [18].

Different Definitions of the Metaphor

As Cunningham first introduced the debt metaphor, he defined it as trade-offs in software quality taken due to conflicting business objectives [1]. In other words, TD is often the result of a compromise between conflicting interests, where software quality is traded off for other goals [9]. Brown et al. name this an "optimization problem", where the goal is to find a balance between short-term and long-term goals [9]. In this case the debt is mostly intentional. But it is not limited to this, TD can also be caused unintentionally, for instance due to inexperience or changing requirements [5, 18].

Today, the metaphor has been developed and extended even further as a result of nearly two decades of ongoing research and several scientific articles have been trying to put TD into words. It has been described as non-optimal solutions during the development of software projects that introduce an ongoing cost [21]. Besides that, it has been expressed as structural quality problems in production code that need to be addressed by the development team [5]. It has been simply defined as a defect in any artifact that is a part of the software system [10] or as deferred investment opportunities [22]. Kruchten et al. specify it as an approach or a design that creates a context, where a unit of work would cost less apriori than aposteriori [3]. It is evident that there exist multiple ways of describing the same metaphor, some more vague than others and this leaves plenty of room for interpretation.

One aspect, that all of these aforementioned definitions share, is that TD comes with a cost. This cost can affect all of the stakeholders of the software system and often results in high maintenance costs, usability problems, confusion, increased development costs, slowed progress, low efficiency, limited scalability, security vulnerabilities etc. [2, 17, 5]. The impact does not only affect the quality of the system and the productivity of the team, but may also have an effect on other dimensions, such as the morale [19].

If developers have to deal with negative consequences of TD on a daily basis, then it is likely to lower their morale and the willingness to produce high quality software will decline as well [19]. From this can be concluded that the costs of TD sometimes tend to increase and accumulate with time and if there is no clear strategic reason or trade-off to take on this debt and no feasible repayment strategy, then one should probably reconsider if it is worth following this path [23].

Interest and Principal of Technical Debt

When talking about the costs of TD, they are often grouped under the term "interest of TD", which was first introduced by Cunningham [1]. The interest of TD is defined as continuous ongoing costs in software development that would not exist, if the software system would be in its "ideal" state, without the TD [5, 10, 19, 12]. It also includes the additional cost of paying back the debt later, as compared to an earlier point of time [12]. It is common for the interest to increase with time and in its later stages can become so high that the whole development comes to a stand-still [18]. As soon as the interest payments become too high, it is time to pay back the principal of TD, i.e. eliminate the TD.

The principal of TD is defined as the cost of the amount of activities that have to be completed in order to completely remove the debt from the software system [5, 9, 12]. The removal typically consists of refactoring and re-architecturing activities, depending on the type of the debt. The main difference between the interest and principal is that the interest has to be paid continuously throughout the life cycle of the debt, whilst the principal only has to be paid once at the end of the debt's life cycle.

Risks and Benefits of Technical Debt

The moment where the total summative interest and principal of TD become equal is also called the "breaking-point" and means that the benefits the TD might have had before vanish [24]. Before reaching the breaking-point, when the total interest is significantly lower than the principal, it often does not make sense to pay back the debt [18]. The reasoning behind this can be compared to deciding, if it makes sense to buy a monthly train ticket or a bunch of one-way tickets. As long as the required amount of one-way tickets is lower than the monthly ticket, it does not make sense to buy the latter one. But a factor that complicates this problem is that the actual principal and interest rate of TD can often only be determined after the TD has already been paid back and they are generally based on inaccurate estimations [9, 18, 3]. As their definitions indicate, interest and principal can only be defined, if the "ideal" and current state of the system are known and the specification of those states is unfortunately not trivial.

Another aspect, common to various TD definitions, is that TD comes with a risk attached to it. This risk is often noted under the name "probability of interest" and

represents the possibility that the negative impacts of TD may or may not materialize in the future [5, 10, 18, 12]. Kruchten et al. state the importance of discussing the future in the context of TD in their paper: "Only when the future is known can technical debt be given an absolute value in terms of effort (to repay)" [3]. In essence, this probability decides, if it is a good or a bad investment to take on TD and at what point of time it should be either reduced or paid back. If it is foreseeable that the instance of TD will reach its breaking-point in the future, then the participants involved should seriously consider whether it is worth taking on this debt [24].

Taking on TD may be vital to success in a software project and can also result in many positive outcomes, just as financial debt [17, 9, 18]. For instance, reducing time-to-market in order to quickly gain a market share, meeting deadlines, or to promptly receive important customer feedback on some novel features [18, 3, 22, 6]. If it is known that a prototype is only used for quick experiments and will be thrown away later, then it does not make sense to invest a lot of time into achieving a very high level of software quality. In this example, the principal of TD is very high, but the probability of the interest having to be paid later is almost zero and therefore it makes sense to take the risk. Thus, as long as the interest and interest probability of TD are low, there is a valid reason to consider trading off software quality against other objectives. Interest, principal, and probability of interest are the three main components that are talked about, when assessing TD and are present in a large number of the publications in the field.

The Model of Technical Debt

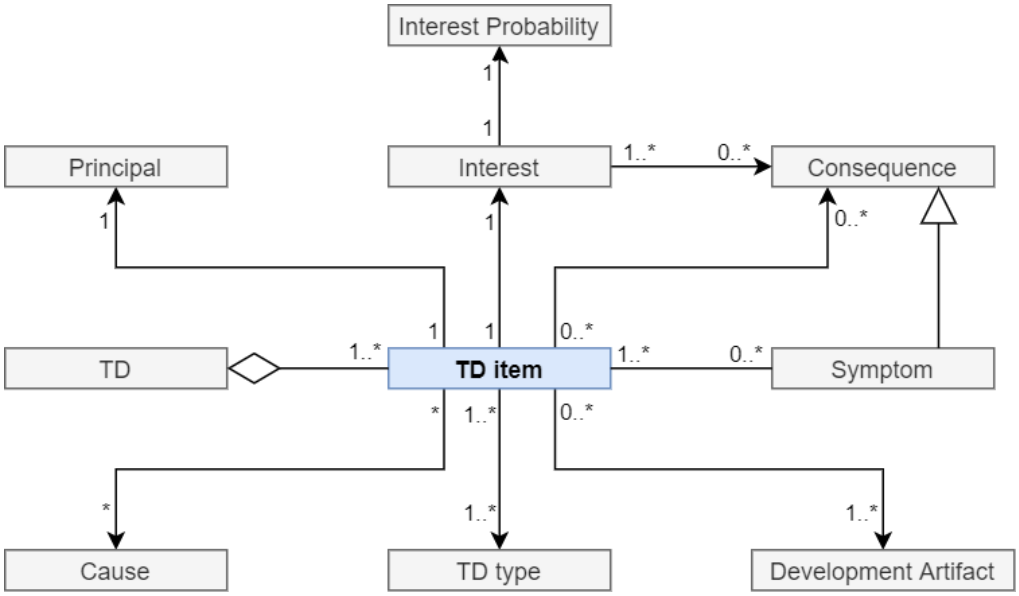

Figure 2.1.: The model of TD [25]. TD in a system is represented as a set of TD items.

In a seminar report Ozkaya et al. propose a conceptual model for the TD metaphor (see Figure 2.1) [25], which is taken as a basis in this thesis. The model represents TD in a software system as a set of TD items, that are associated with at least one tangible development artifact, for instance the code, tests, documentation, or a defect [25]. Each TD item can have multiple causes and symptoms that lead to consequences, such as quality issues, project slippage, or increased costs [25]. The list of TD items from practice, that forms the footing for the analysis phase of this thesis, is partly structured based on this model.

Technical and Financial Debt

As one probably has already noticed, the components of the TD metaphor have been borrowed and adapted from the financial world, where the term "debt" has been known for a much longer period of time. This analogy is deliberate, as it makes it much easier to explain software quality issues and the need for refactoring to stakeholders, who are often not that familiar with the world of engineering [3, 19, 4]. However, it must be borne in mind that TD is not exactly the same as financial debt [10]. An important difference is that neither the principal, nor the interest of TD might ever have to be paid back [10, 18], without it resulting in a bankruptcy. As mentioned earlier, despite being present in a system, TD might not cause any pain and can remain hidden. A further significant difference is that TD might be taken on unintentionally, whilst financial debt always has to be taken deliberately [9]. Even if developers take time to plan ahead and keep an eye on software quality, TD can occur due to external factors, like for instance changing requirements [9]. In addition to that, compared to financial debt, TD is much more difficult to measure and to define [20]. It is often challenging to estimate the interest rate and principal of TD, as their definition depends on the determination of an optimal state for the system [20]. Besides that, estimating the impact also hinges on the probability of interest and future developments [20]. These differences and limitations have to be kept in mind, when using the TD metaphor to discuss software development issues with non-technical stakeholders.

2.3. Types of Technical Debt

The TD metaphor has come to stay and the number of publications concerning the topic is growing by each year. Practitioners and researchers have taken time to map the ontology of TD and the fact that it still continues to rise as a topic shows that this task is more complicated than one would expect at first sight. TD can be categorized based on various dimensions, for instance, based on the artifact, where it resides or based on its causes and effects. Defining a common ontology is important, as this forms a basis for future research and discussions. Furthermore, a categorization makes it possible to map typical characteristics of certain types of debt and this supports identification and

management activities.

A commonly cited categorization of TD has been specified by Martin Fowler. In his so-called TD quadrant, he divides TD between the "reckless/prudent" and "deliberate/inadvertent" dimensions (see Figure 2.2) [26] and takes a look at it from the perspective of causes. In the case of reckless and deliberate TD, the developers are aware of going into debt and they do not have a valid strategic reason for doing so. Most software engineers are aware of the seven deadly sins of software engineering [27]. This type of TD is best described as "apathy", i.e. not bothering to fix known problems. Reckless and inadvertent TD in turn is best described as "ignorance" and is introduced when developers do not even care about understanding the underlying problem. On the other side of the quadrant we have prudent and deliberate TD. This type of TD is taken on, when developers have to make trade-offs in software quality due to other factors of higher importance. Developers are aware of the TD and the trade-off is often strategic or tactical, acting as a leverage [3, 19], e.g. taken to reduce time-to-market. And finally we have prudent inadvertent debt that is accrued due to the natural process of learning. Developers will notice in hindsight that their past decisions were not optimal and led to TD. The latter two types of TD can be seen as being more positive, because developers are at least noticing and admitting the problem, which is the first step in solving and managing TD. Most of the TD in practice falls into the category of prudent and inadvertent debt [4] and this shows that it is often difficult to anticipate debt and that its existence will only become clear later.

Tom et al. similarly distinguish between strategic, tactical, incremental, and inadvertent TD [19]. Strategic and tactical debt both fall into the category of prudent and deliberate debt, former taken as a long-term leverage and latter taken as a short-term leverage, for instance in order to complete a task before a deadline [19]. Incremental debt is accrued in small steps due to ignorance of good software engineering practices, like for example method length and commenting [19], and inadvertent debt simply taken on unintentionally, for instance due to the process of learning [19].

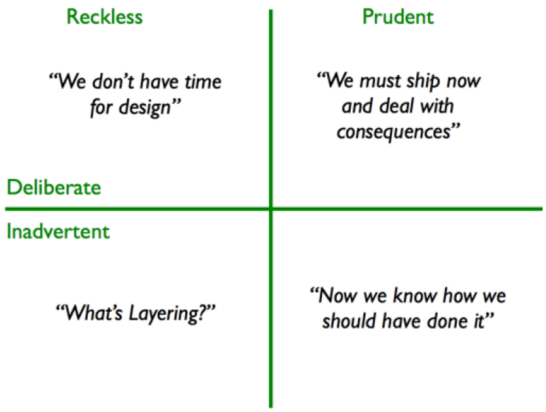

Figure 2.2.: The TD quadrant [26].

Other ways of categorizing TD, are based on the artifact or the layer of the software system, where it occurs; on its characteristics; or on the quality gap that it represents. Based on reviewed literature, we identified the following types of TD:

TD Type	Definition
Design Debt	TD that is located in the source code or micro-architecture of the system and typically caused by the misuse or ignorance of software patterns [2, 10, 28, 29, 12]. Typical examples of design debt are low cohesion, high coupling, and violations of the principles of good object-oriented design [12]. Can be caused by the use of antipatterns or by the use of patterns that become suboptimal with time, as requirements change [19].
Architectural Debt	TD that is located in the higher-level architecture of the system and requires re-architecturing activities for elimination [10, 28, 29, 12]. Typical examples of this debt are monolithic legacy systems and modularity violations [28]. Like design debt, so can architectural debt be caused by either antipatterns or by architectural solutions that decay with time [19, 12].
Knowledge Distribution/Documentation Debt	Described as missing, erroneous or incomplete documentation of the system, e.g. lack of code comments [10, 28, 19, 29, 12].
Self-admitted Debt	A more abstract type of TD that has been documented in source code comments, i.e. cases where developers introduce new debt, are aware of it and leave comments about it, for instance in the form of "TODO's" or "FIXME's" [21].
Testing/Test Debt	Deficiencies in testing activities or artifacts, for instance low test coverage, missing integration and system tests, or not executing test suites [10, 28, 19, 29, 12].
Test Automation Debt	A subtype of testing debt that results from the failure to automate manual test suites [10, 28, 19, 12], e.g. manual user interface (UI) tests that could be substituted by automatic UI testing tools.
Code Debt	Like design debt, this type of TD is located in the source code of the system, but is even more low-level and caused by the use of bad coding practices, for instance the use of long and complex methods; duplication; or not using any code indentation [10, 28, 19, 29, 12]. Code debt often induces maintainability issues [12] and can be eliminated with the help of code refactoring activities [19].

Defect Debt	Any defects or bugs that have been identified by a stakeholder of the system and are usually tracked in a ticketing system [10, 28, 29, 12]. There is no consensus among researchers, if defects should be a type of TD. For instance, Kruchten et al. believe that low external quality, that is visible to the user in form of defects, should not be considered as TD [3].
Requirements Debt	TD that results from requirements that are known to the development team, but have not been fully or correctly implemented [10, 28, 29, 12]. Can also be described as the difference between the requirements specification and the actual implementation of the system [12]. Again, Kruchten et al. suggest that unfinished and postponed tasks and features should not be considered as TD [3].
Environmental Debt	TD that is caused by the environment or the context of the software system, for instance external processes, hardware, infrastructure, or neighboring systems [22, 19].
Infrastructure Debt	A subtype of environmental debt that is caused by the infrastructure, used by a software system and usually hinders development activities and lowers the overall quality of the system [10, 28, 29, 12]. For example, data loss caused by the use of a non-redundant database server.
Social/People Debt	A subtype of environmental debt that is caused by inefficient social structures and organizational forms that have a negative impact on the software system [10, 28, 30, 12]. For example, wrong team size, bad knowledge distribution, or lack of team members with necessary expertise. It can again be debated, if these issues should be considered to be a type of TD.
Process Debt	A subtype of environmental debt, resulting from software development processes that are inefficient or have become obsolete [10, 28, 31, 12]. For example, the use of the Waterfall model in a project, where agile methodologies would be much more appropriate.
Build Debt	A subtype of environmental debt, in the form of slow or nonoptimal build processes [10, 28, 29, 12], e.g. build times that make continuous integration difficult.
Versioning Debt	A subtype of environmental debt, that resides in the versioning schemes and processes of a software system [10, 29, 12], e.g. the use of SVN in a case, where Git would be appropriate; or unnecessary code forks [12].
Service Debt	TD common for service-oriented architectures, where the system contains inefficient or useless services that lead to a service-landscape, which is not fulfilling the requirements of the application [10, 28, 12].

Usability Debt	TD that affects the usability of the system and results from interface designs that are illogical or impractical [10, 12]. For instance, a website that uses an inconsistent style for buttons.
Enterprise Architecture (EA) Debt	EA debt takes a higher-level view onto the software system and also incorporates the business perspective [17, 32]. Hacks et al. define it as: "a metric that depicts the deviation of the currently present state of an enterprise from a hypothetical ideal state" [17]. EA debt often results in organizational bottlenecks, communication problems, and conflicting objectives, that have a negative impact both on the software system as well as on the organization [17]. Compared to other types of TD that take a system-wide perspective, EA debt takes a rather organization-wide perspective.
Contagious Debt	A more abstract type of TD mentioned by Besker et al, that tends to spread like a virus and accumulates with time [11]. It follows the broken window theory, mentioned by Andrew Hunt and David Thomas in the book "The Pragmatic Programmer" [33]. The idea behind it is that software issues often force or guide developers to introduce even more issues, if not taken care of. For example, a broken high-level architecture might hinder developers from implementing new features optimally and can lead to semi-correct solutions.

Table 2.1.: Types of TD and their definitions from literature.

The aforementioned types of TD have overlaps between each-other [10], look at debt from different dimensions, and some of them are simply subtypes of a more abstract type of debt. In addition to that, different types of debts are not independent from each-other and one can be a cause for another type of debt. For example, architectural debt can give rise to usability debt, when an incorrectly integrated payment system makes users jump illogically between different portal pages and frames. The infamous Conway's law [34] is a further example of this, where social debt or EA debt enforces architectural debt, as the architecture of a software system starts mirroring suboptimal communication structures of an organization. It is also interesting to note that some researchers have a more liberal way of looking at TD, whilst some others are conservative and claim that certain types of issues should not be considered as TD. Despite this, the ontology provides a useful classification scheme and vocabulary for discussing TD.

Even though this list of different types of TD already seems quite thorough, we have identified a gap in it and this will be explained in the following section.

2.4. Previous Work on Domain Level Technical Debt

Several research articles state that most of the literature about TD mainly deals with implementation level or source-code-related TD and a considerably smaller part of it reports about architectural or domain-oriented debt [2, 10, 11, 12]. In fact, a mapping study from 2015 found that 78% of the articles focus on TD at the lower technological level and only 2% of them examine higher-level design issues [2, 10]. A reason for this could be that low-level TD can often be found by looking directly at the source code, supported by various code analysis tools, and this makes its analysis and management more straightforward [10, 12]. However, practice shows that the under-investigated high-level TD, such as architectural debt, often has much more severe negative impacts on the software system [2]. It is more difficult to fix the high-level design of the system, compared to performing lower level code refactoring. This trend can be noted as a gap in research.

In 2019 Harald Störrle and Marcus Ciolkowski, aware of this gap, published a paper about a new type of high-level TD that they named "Domain Level TD" [2]. To the best of our knowledge, domain level TD or simply domain debt has not been covered sufficiently by existing research. They define it as "the misrepresentation of the application domain by an actual system" and claim that it is a practical notion for discussing trade-offs between business objectives and the high-level design of the system [2].

Characteristics of Domain Level Technical Debt

Compared to implementation level debt, domain debt cannot be identified by simply analyzing the source code [2]. What makes it special is that domain knowledge and expertise are essential for being able to identify and manage domain debt [2]. Typical artifacts that are useful, when analyzing domain level TD, are the documentation, macro-architecture, requirements, models, and sub-systems [2]. The problem with these artifacts is that they are often non-existent, tacit, or incomplete, making it almost impossible to automate the detection of domain debt [2]. In addition to that, not every team member has sufficient knowledge to be able to take a look at the system from such a high-level perspective. Team members, aware of relevant past decisions, have often either left the project, have simply forgotten the information, or have not been able to track all of it throughout the history of the project.

As explained earlier in this thesis, in order to identify and manage TD, it is first necessary to know the current state of the system and map the debt, and secondly determine and compare it to alternative "ideal" state(s). The aforementioned reasons, missing artifacts and need for domain expertise, make both of these steps significantly more difficult and expensive for the case of domain debt [2]. What makes this problem even worse, is that domain debt tends to cause more serious consequences than implementation level debt and this pain is often not only felt by the development team,

but also other stakeholders, such as the users of the system, as it commonly results in usability issues [2].

Adding to the problem, domain debt and implementation level debt are often independent from each other, meaning that it may be invisible to the developers, as there are no indicators for it in the code [2]. Meanwhile the users and other stakeholders may be dealing with severe issues and if feedback cycles do not work, then it may remain hidden for longer times.

A further characteristic of domain debt is that it usually crosses the borders of a system and also affects other systems and business units, ultimately leading to EA debt, organization-wide inefficiencies, misunderstandings, and other issues [2]. This high concentration of stakeholders means that the removal of domain debt often requires wider collaboration than is typical for other types of TD, resulting in higher management overheads [2].

Causes of Domain Level Technical Debt

Störrle and Ciolkowski mention four typical reasons for why domain debt is born. Firstly, it might result from decaying design [2], when changes made to the system do not follow and break the initially planned design. As long as no time is spent on updating the domain level design, every minor modification cumulatively adds some domain debt to the overall load.

Secondly, it may be caused by the process of learning [2], similarly as for prudent inadvertent TD, explained earlier in this thesis. Developers gain new domain knowledge during the process of development and this changes the understanding of the optimal domain level design of the system.

Thirdly, domain debt can be caused by external priorities that change during the time of the development and result in half-finished features [2]. This case is especially characteristic for projects that use an iterative workflow and where the development of a feature often has to be broken down into multiple increments [2]. The strength of agile practices - reacting to changing requirements - might also turn into its weakness, when plans keep changing between increments and long-spanning features are not implemented holistically. Let us picture an architect, who first sets off by designing a football stadium, but midway, due to a change in requirements, starts designing a concert hall. Because of cost and time constraints he continues using the same project. The result is a hybrid, that would not fulfill any of the two use cases properly. The same can happen in software projects that face constant change.

Fourthly, domain debt can result from a domain or a set of constraints, that shift later in time, after the system has already been completed [2]. A design that is fit for its purpose today will probably not be so in ten years time due to the evolution of requirements, as also manifested in the first law of Lehman [16]. Therefore, if the

domain level design of the system does not evolve over time, it soon starts accruing debt.

To summarize, the only ways of avoiding and reducing domain debt are to have a thorough planning phase at the beginning of the project, that is based on a sufficient amount of domain knowledge and to keep updating the domain level design throughout the life cycle of the system [2]. The problem with this is that it conflicts with a trend that has become very common nowadays - agile methodologies [2]. Agile frameworks often do not support profound upfront design and they also limit domain documentation that is required to assess and manage domain debt. Thus, the goal is to find a balance between agile practices and activities that involve designing, planning, and documenting.

Examples of Domain Level Technical Debt

The concept of domain level TD is best clarified with an example. The recent struggle around developing smartphone applications for alerting people about Covid-19 cases and contacts. In the European Union, many countries quickly started developing their own applications, instead of taking some time for designing a joint solution. Some countries, like for instance the United Kingdom, started pursuing a centralized model, where data, collected by smartphones via Bluetooth, is sent to a central server for analysis [35]. Other countries, like for instance Germany, took a decentralized approach, where contact matching and risk analysis is performed directly on the smartphone of the user [35]. These different approaches, among many other differences, make it very hard to integrate this heterogeneous landscape of Corona applications. Yet the domain includes a use case that goes beyond the borders of one country. In the European Union, where borders have become symbolic and economic and social ties are very strong, people are used to travelling around a lot. This implies that it would make sense to have a European wide Corona application or at least to be able to exchange data between the applications of different EU member states. The fact that a well-integrated European wide solution does not exist, can be seen as domain debt. The interest is paid by the users of the applications, who are not able to use the same application in different countries and are not being notified about infected foreigners; and by the governments and medical organizations, who are not able to track and reduce infections resulting from international travel. At the end of the summer season in 2020 the Robert-Koch-Institut reported that around 31% of the Covid-19 infections in Germany resulted from international travel [36]. What distinguishes this case of TD from typical implementation level debt, is that it does not necessarily affect the developers of the application and does not slow down the development process. The Corona application itself might meet the highest software quality standards, but is nevertheless incomplete.

Based on the previous example we can notice some significant characteristics of domain debt. Namely, compared to typical TD, it does not always primarily affect the developers of the system, but often also other stakeholders, like for example the users.

The presence of domain debt does not definitely mean that the codebase of the system has a low quality, but rather that the domain level design is flawed and the system cannot fulfill all its use cases and requirements.

Although in the example, domain debt is present at higher abstraction levels of the system, Störrle and Ciolkowski mention in their paper that this is not always the case for it [2]. Domain debt can also be caused by smaller low-level units, for instance by poor entity or function names [2]. They illustrate this with an example, where the naming of data entities is inconsistent in a Repair and Research system for vehicles and this results in frequent misunderstandings that affect both the users of the system as well as the developers [2]. In contrast to the previous example, here also implementation level artifacts, such as the source code and database structures, play an important role and the confusion slows down development activities.

Domain Debt and other Types of Technical Debt

In the previous chapter, we listed types of TD, that have been mentioned in different scientific publications. None of the types in that list is capable of fully covering instances of debt that fit to the definition of domain level TD and this affirms the existence of a research gap. Possible candidates for covering domain debt could be requirements debt, environmental debt, usability debt, and EA debt. For the remaining types it is by definition relatively clear that they deal with completely different aspects or artifacts of the system, for example code debt, design debt, testing debt, test automation debt, and architectural debt tend to only concentrate on the implementation level design of the system.

Requirements debt does not cover domain debt, because it is limited to unimplemented features, that by definition are also known to the development team. Domain debt however could also result from features that have already been implemented, such as in the example with inconsistent naming of data entities, or from factors that are not known to the development team.

Environmental debt does not cover domain debt, as it only concentrates on the external context of the system, consisting of various elements and processes. Domain debt is not solely limited to external elements and takes a more high-level holistic view of the system into account, which also considers internal elements. In other words, environmental debt describes a quality gap in the external context of the system, but domain debt describes a quality gap between the external context and the system itself.

Usability debt does not cover domain debt, as it focuses on the design of interfaces. An illogical interface however does not necessarily mean that the domain of the system has been misrepresented and that the system cannot fulfill a certain use case.

Finally, EA debt takes an organization-wide perspective and also considers organizational inefficiencies, whereas domain debt rather takes a system-wide perspective.

Use of Domain Level Technical Debt

But why should domain debt be yet another type of TD? One of the main ideas behind the introduction of the debt metaphor into software engineering is to use it as a way of communicating about software development issues. Also domain-related matters are a part of software development and should be discussed with other stakeholders. In this case, the term is not only useful for explaining domain issues to non-technical managers, but also to team members who are only familiar with lower technical levels. It is a well-known problem in software engineering that the overall purpose of the system is sometimes misunderstood by developers due to a lack of domain knowledge.

Keeping the scope of TD only limited to implementation level and code-related issues will reduce its usefulness, since software systems cross different levels of abstraction and include various artifacts. Even more so as we mentioned earlier, domain debt is not only related to one particular level of abstraction, but can for example also be found in implementation level artifacts. Thus, it is important to take a holistic view into account, when talking about TD. Software artifacts are not independent little islands, but have dependencies between each other. Smaller code-related issues might actually be caused by something that went wrong in another part of the system. Domain debt represents a quality gap that is not covered by existing types of TD.

There are multiple facades to domain level TD and the goal of this thesis is to further present, describe and analyze cases of it from practice and to map and document their characteristics and effects. These results then serve as ingredients for working on a crisper definition for domain debt that helps increase the understanding of it.

2.5. Identification and Management of Technical Debt

The management life cycle of TD consists of the following main three activities: (1) identification; (2) estimation and measurement; (3) prioritization, tracking and payment [2, 22, 4, 12]. Apparently, these aforementioned activities are non-trivial and no commonly accepted standard procedures exist for conducting them [3].

Identification of Technical Debt

Like with most problems in life, in order to deal with them, they first need to be identified and acknowledged and this is not any different for TD. A lot of effort has been made to automate this step, but research has shown that software analysis tools are far from sufficient and can only help in detecting certain types of debt [22, 4, 13]. The main reason behind this has been discussed earlier in this thesis and lies in the fact that TD cannot always be directly identified based on the source code, which is typically used by modern analysis tools. Many other artifacts, contextual information, and domain

knowledge are often required to be able to elicit TD and current software analysis tools are often not able to take those elements into account. One reason for this is that a large part of this information and knowledge is usually tacit and undocumented. For example, a static software analysis tool would not be able to determine, if the system fulfills a certain use case optimally. Nevertheless, tools can be helpful when identifying TD that relates to the implementation level quality of a software system, especially defect debt [13].

Tools should best be used in combination with manual reviews, performed by software developers and domain experts. Namely, research shows that there is usually only a small overlap between TD identified by automated tools and manual reviews [21, 13], thus combining them makes sense. In addition to that, there are also large differences between the sets of TD items identified by different developers, so aggregating them is more reasonable than trying to form a consensus [13]. Even though manual reviews take more time and effort, they have several clear advantages over their automated counterparts. Firstly, they are generally more accurate and tend to identify TD items that have a larger negative impact [4, 13], i.e. they filter out smaller irrelevant defects and false positives. And secondly, they are capable of taking contextual and historical information into account [13], which for instance is necessary for being able to detect domain debt.

TD management is oftentimes reactive and becomes relevant once the debt's effects become visible or cross a certain pain threshold [4]. Ernst et al. ascertained in their study, that around 27% of software practitioners claim to fail first and deal with TD in hindsight [4]. Therefore, the identification is often based on effects and TD indicators that have become visible with time. Rios et al. found that in almost a quarter of cases, these effects originate from the category of software quality issues, being low quality, low performance, bad code etc. [37]. Alves et al. further confirm this tendency in their study, where they state that the most cited TD indicators are code smells and that most of the research concentrates on identifying TD from source code [10]. This implies that even though manual reviews are also capable of detecting more abstract types of debt, they also tend to fall into the same pitfall trap as automated tools and carry horse blinders that only focus on source code. Therefore, the TD identification step should utilize a wide variety of strategies that take different software artifacts and abstraction levels into account [10]. A core part of this thesis is to elicit indicators and characteristics of domain level TD, that could then be used in the identification stage.

Estimation and Measurement of Technical Debt

In the second step of the management process, the debt should be estimated and measured by determining the principal, interest, interest probability, and other factors of the TD item. This step is very challenging, as it is often impossible to determine impacts of the debt and outcomes of refactoring activities, even after they have already been

completed [9, 6]. Therefore, this phase is prone to subjectivity and the principal, interest, and its probability are often estimated based on historical defect data [13].

The principal can be determined as the number of problems that have to be fixed and the time and cost that is necessary for fixing them [24]. The interest can be measured by comparing the effort of maintaining and using the status quo system versus the refactored system [24]. It is important to take a broader view and measure the accumulated total cost of debt and not to handle each smaller part of it separately [22, 6]. Estimating the interest probability is even harder and sometimes comparable to gambling in a casino or investing on the stock market. Its estimation depends on the possibility that certain events that lead to interest payments take place in the future. Therefore, those measures should be enjoyed with caution, but can provide a way of prioritizing and discussing the items of debt.

It must be kept in mind that one of the central ideas behind the TD metaphor is to use it as a medium for explaining software development issues to non-technical stakeholders. Hence, a sole list of TD items is not sufficient to fulfill this purpose and estimates, preferably in monetary value, are an essential part of TD management [4]. After estimation, financial approaches, such as Net Present Value [22], Cost/Benefit [24], Lost Value to Customer [20] etc. can be used to measure and determine the overall impact and priority of the TD item. In general, items with a high interest rate and low principal have the most substantial impact and should be considered for refactoring [24].

Prioritization and Tracking of Technical Debt

The prioritization and tracking step involves adding the TD items to a backlog list, prioritizing and ordering the list based on the measured impacts, and managing them over a ticket tracking system [22, 6, 13]. As Ernst et al. claim: “Technical debt remains something that at best is managed with identifying tags on the issue tracker” [4]. All debt-related data should be documented in this list, including the causes, effects, descriptions, estimates, measures and other attributes [13].

The list of TD helps monitor the health of the software system and visualizes the difference between the envisioned optimal design and the current state [4]. It must be made understandable and available to all stakeholders, also non-technical, who are in the position to steer the project and the development activities [6]. The goal of the list is to facilitate a dialogue between managers and developers and to find the best compromise that balances business goals and technical risks [6].

Also in this thesis, the basis for the data analysis is formed by a list of TD items, that to a large extent have been estimated, prioritized, and tracked in a ticketing system. The ticketing system also helps track how the debt has evolved over time and makes it possible to compare estimates to actual efforts in a retrospective analysis.

Managing Technical Debt in Agile Projects

Earlier in this thesis we discussed that agile methodologies often conflict with approaches that can be used to minimize impacts of TD, such as thorough planning and design phases at the beginning of a software project. Yet there is a way to make use of agile peculiarities and combine them with TD management strategies.

Noopur Davis provided empirical evidence in a study that TD can be effectively managed by using multi-level Definition of Done's (DOD) in agile projects [38]. A DOD is a decision gate that contains criteria that have to be fulfilled and a list of items that need to be completed before a process stage can be completed [39]. Davis suggests that such DODs should be used between all stages of agile development, such as stories, sprints etc. and should contain actions and requirements that help enforce high quality and minimize the amount of TD [38]. This means that TD identification and management activities would be a natural part of agile development and they could be performed often and in iterations. This would reduce the cost of defects and other debts, as the time between their introduction and removal could be minimized [38].

3. Research Methodology

This chapter provides an overview of the research methodology used in this thesis. There are two sources of data: firstly, a list of TD items, collected from a project from the industry; and secondly, case studies on instances of domain level TD, conducted with software experts. The Research Questions listed in section 1.1 will be answered based on these two sources of data. RQ1 and RQ2 will be answered based on the domain level TD case studies and RQ3 will be answered based on the list of TD items. The following sections describe the overall research plan and the two trains of data collection.

3.1. Research Plan

The main part of the research was conducted between November 2020 and March 2021 in collaboration with the software company QAware GmbH. The research plan consists of the following steps:

1. Data Collection Phase:
    a) Collecting a list of items of TD from a candidate project from the industry. Further described in section 3.3.
    b) Collecting items of domain level TD and describing them in a detailed, yet confidential manner as case studies. Further described in section 3.2.
2. Analysis Phase:
    a) Identifying characteristics of domain level TD in practice based on the case studies and previous literature. Abstracting the characteristics into a generally applicable procedure for domain level TD identification, e.g. as a check list.
    b) Sanitizing the assessment of the candidate project to create a publishable list of issues. Identifying the economic and technological impact of different levels of TD based on the list of TD items.
3. Validation Phase:
    a) Validating the domain level TD identification procedure, by conducting a Delphi study involving senior industry experts from QAware GmbH. By comparing different projects and contrasting the findings with experts perception, we expect to achieve a considerable degree of generalizability. Further described in subsection 3.2.2.

3.1.1. Case Study Research

The overarching approach that forms the backbone of this thesis is Case Study Research [40]. By definition, case studies are empirical inquiries that explore smaller parts of a software engineering phenomenon, based on an extensive set of data and evidence sources [40]. Performing case studies is an effective way of gaining more insight into a phenomenon in a real-life environment and achieving a better understanding of it [40].

The case studies in this thesis are based on items of TD amassed in several software projects from the industry, which are enriched by background information collected via expert interviews. Case studies are suitable for research, where lack of control is not a critical issue and it is more important to have a real-life context and a high degree of realism [40].

The research questions and case studies in this thesis are mainly of exploratory nature and utilize a mixed methods approach. Exploratory research aims at discovering and mapping new aspects of a phenomenon and forming new ideas and hypotheses [40]. Exploratory research questions typically answer questions that begin with "what" [40] and this is also the case for two out of three research questions in this thesis. In general, exploratory research fits well together with a case study strategy [40].

A mixed methods approach analyzes both qualitative and quantitative data, whereby quantitative data is analysed using statistics and qualitative data using categorization and sorting [40]. In this thesis the list of TD items is analyzed statistically, when determining the number and size of issues and their economic and technological impact. Eliciting and describing domain debt characteristics and effects is thereby rather qualitative.

The Case Study Approach consists of five main steps [40]: (1) designing the case studies, defining its goals and research questions; (2) preparing for data collection by defining procedures for it; (3) collecting the data; (4) analyzing the data; and (5) reporting the results of the study.

3.2. Domain Level Technical Debt Case Studies (RQ1, RQ2)

The domain level TD case studies were used to answer the Research Questions RQ1 and RQ2, described in section 1.1.

Six case studies were conducted with two senior software consultants at the software company QAware GmbH. Both of them specialize on eliciting and managing requirements and on the domain-oriented design of software systems. The items of debt, described in the case studies, were selected from several different software systems and projects. Some of the items were selected from the list of TD items, described in section 3.3. When selecting the items of debt, it was important that they have aspects that fall under the definition of domain level TD. Both of the interviewees were familiar

with the concept of domain level TD.

The interviewees filled out a form with a total of 34 questions (see section A.1) that were compiled based on previous literature. The purpose of the free-text questions was to cover as many different aspects as possible related to the life cycle of the debt.

The objective of the questions 1-14 was to obtain contextual information about the system, the stakeholders, and the project. Questions 15-26 focused on the item of debt itself, its identification, and causes. Questions 27-32 were targeting the effects and consequences of the debt and questions 33-34 were directed at the management of the debt. In addition, several conversations with the interviewees were conducted in iterations to clarify missing information, and the interviewees checked the final content of the case studies and provided feedback to avoid and fix misunderstandings and errors.

Where possible, data from issue tracking software Jira was used to quantify the interest and principal of the items of debt. At QAware GmbH, a T-shirt-based scheme is used. The T-shirt sizes represent effort ranges and are used to express the cost of tasks and issues. The exact calculation cannot be published for non-disclosure reasons. For the purpose of this thesis, we use an approximate and rough mapping to effort, as visualized in Table 3.1. A person-day (PD) of work is equal to 8h.

T-Shirt	Person-Day (PD)
XS	$\leq$ 0,5 PD
S	$\leq$ 1 PD
M	$\leq$ 2,5 PD
L	$\leq$ 4 PD
XL	$\leq$ 7 PD

Table 3.1.: The T-shirt-based task size scheme at QAware GmbH.

A list of characteristics was selected based on the observations made from the case studies and on information gathered from previous literature. The case studies were compared to each-other based on these characteristics. The objective of this step was to answer RQ1, by mapping common effects and characteristics of domain level TD in practice.

The results of the analysis of the case studies were then used to answer RQ2 and put together a general checklist for identifying domain level TD. The questions in the checklist were categorized and weighted based on experiments with items of TD from both the case studies as well as from the list of TD items. The checklist and specifically its questions and wording were then evaluated during the validation phase by senior software engineers at QAware GmbH. The structure of the validation process is described in subsection 3.2.2.

3.2.1. Structure of the Case Study

The structure of the case studies consists of three main parts: (1) an overview of the problem and its context, (2) classification and observation data, and (3) general observations.

The first part explains the context of the problem, including the business context, relevant domain knowledge, and technical terms. After that, the content of the problem itself is explained. The life cycle of the debt is further illustrated by an image. The components of the image are explained in Figure 1.1.

	Represents the **domain** of the system. Its size depicts the scope and focus of the domain. Ideally, the system should be located inside the domain.
	Represents the **system** in question.
	Represents the movement or change of the **domain**. The text label names the action that led to it.
	Represents the movement or change of the **system**. The text label names the action that led to it.

Figure 3.1.: An explanation of the components of the images, illustrating the life cycle of domain level TD.

The second part classifies the debt and provides more detailed observation data. It is broken down into the following subsections:

- **Criticality and Properties of the Debt** - the criticality of the debt is rated on a three-point scale from “low”, “medium” to “high”.
  - “low” - the cost of the debt is considerably low and the project can survive with it for longer times.
  - “medium” - the debt is causing a remarkable cost and should be repaid in the coming years.
  - “high” - if the debt is not repaid right away, the cost will ultimately lead to the failure of the project.

The properties describe the general nature of the debt, namely its distribution and location in the system and its contagiousness.

- **Principal of the Debt** - quantitatively and qualitatively describes the principal of the debt.
- **Interest of the Debt** - quantitatively and qualitatively describes the interest of the debt.
- **Interest Payers** - describes what stakeholders pay the interest of the debt.
- **Causes of the Debt** - describes the reasons and mechanisms that led to the debt.
- **Location in the TD Quadrant** - places the debt into the TD quadrant [LL4].
- **Management of the Debt** - describes how the debt has been managed in the project. Specifically, what decisions have been made and whether the debt has been repaid or not.
- **Relevance of Domain Knowledge** - classifies and links the debt to domain level TD. Explains the importance of domain knowledge in the life cycle of the debt.
- **Other Potential Types of TD** - classifies and links the debt to other potential types of TD.

The third part provides a summary of the general observations that were made based on the case study with regard to domain level TD. The goal is to highlight characteristics of domain level TD in practice and compare them to characteristics that have been noted in previous literature.

3.2.2. Structure of the Validation Process

The checklist and specifically its questions and wording were validated by eight senior software engineers at QAware GmbH. The precondition was that the evaluators were already familiar with the concept of TD. The meaning behind domain level TD was made clear to the evaluators in advance. The checklist was sent to the evaluators and they provided individual feedback.

The goal of the validation was to receive feedback from software experts and to improve, consolidate, and elaborate the checklist. The objective was to clarify if the questions in the checklist are (a) understandable and (b) fulfilling their purpose. The evaluators were tasked to read the questions in the checklist and to paraphrase more complex questions in their own words. Additionally, they were instructed to imagine some items of TD from the backlogs of their projects and to try to apply the checklist.

The following questions were asked from the evaluators:

- How do You understand the questions? If necessary, try to explain more difficult questions in Your own words.

- What do You think of the questions? Are they purposeful or not and why?
- Would You remove or add any questions?
- Do You think the four categories and weights are appropriate and useful?
- Do You think that the checklist could be useful for assessing domain level TD?

The feedback from the evaluators was merged and evaluated and corresponding corrections and improvements were made to the checklist. In a second iteration this updated checklist was again sent to the evaluators for a final round of feedback.

3.3. List of Technical Debt Items (RQ3)

A list of TD items was put together, by surveying a project from QAware GmbH for existing items of TD, estimating their economic and technological impact, categorizing them, and analyzing the findings. Hereinafter the project in question will be referred to as "the Candidate Project".

The objective was to have a comprehensive list of TD items that covers various types of TD. The results of the analysis of the different levels of TD and their impacts was then used to answer RQ3, described in section 1.1.

An initial version of the list of TD items was created, when the Candidate Project was transitioned to QAware GmbH from another company in 2020. The software system was systematically analyzed by four senior software architects in order to assess its condition. This analysis included both manuals reviews, as well as findings received from static software analysis tools, such as Sonar. The list has a high quality and is detailed. Considerable attention was paid to its preparation, as it was the basis for taking responsibility for the system. Miscalculations would have led to financial risks. All available artifacts that are related to the software system were evaluated. The main part of the analysis was performed during the period from February 2020 to the middle of March 2020. Later, minor additions and clarifications were made.

As a part of this study, the list of TD items was supplemented by interviews with a software consultant, who has been involved in the project since its transition to QAware GmbH. Additionally, data from issue tracking software Jira was used to evaluate the principal and interest of the items of TD. The structure of the list is presented in subsection 3.3.1.

Data about the criticality, principal, interest, interest payers, and interest effects of the TD items was used to analyze the economic and technological impact of different levels of TD. First, the observed absolute frequencies of the different dimensions were represented as contingency tables. Second, the data in these tables was grouped and abstracted in order to be able to apply the Chi-Square test for independence. The Chi-Square test can be used if no more than 20% of the expected frequencies are less

than 5 and none are less than 1 [41]. In order to achieve that, different types of TD were grouped into more abstract levels of TD according to similarity; data adjacent on a scale was pooled into a larger group; and parts of the tables, where the data volumes were too small, were excluded from the analysis.

3.3.1. Structure of the List of Technical Debt Items

Each item in the initial unsanitized list of TD items has the following data:

- Origin - the type of artifact from which the debt was discovered, e.g. the code, the documentation, the architecture etc.
- Finding - describes the content of the debt (removed from the sanitized list).
- Action - describes what has to be done to repay the debt (removed from the sanitized list).
- Criticality - the criticality of the debt, rated on a five-point scale from “very low”, “low”, “medium”, “high” to “very high”. Based on the interest probability of the debt.
- Principal - a vague estimate for the principal of the debt, rated on a five-point scale from “very low”, “low”, “medium”, “high” to “very high”.
- Interest - a vague estimate for the interest of the debt, rated on a five-point scale from “very low”, “low”, “medium”, “high” to “very high”.
- Interest Effects - mentions the general aspects that the interest of the debt affects.
  - Productivity - the debt reduces the overall productivity in the project, by slowing down progress.
  - Maintainability - the debt reduces the maintainability of the system, e.g. lack of documentation.
  - Quality - the debt hinders the measurement and assurance of system quality. It prevents the identification and assessment of further items of TD.
  - Stability - the debt reduces the stability of the system, e.g. by causing frequent system outages.
  - Performance - the debt reduces the overall efficiency of the system or of a part of the system.
  - Security - the debt causes security vulnerabilities.
  - Usability - the debt reduces the usability of the system, e.g. certain use cases cannot be serviced properly.
  - Diagnosability - the debt complicates debugging and monitoring tasks, e.g. the absence of proper logging mechanisms.

- Interest Comment - additional explanations for the interest of the debt (removed from the sanitized list).
- Interest Payers - describes what stakeholders pay the interest of the debt.
  - Product Owner - the interest causes business risks for the company and the product owner, e.g. fines related to security issues.
  - Users - the users of the system pay the interest, e.g. low user satisfaction, impediments due to system failure.
  - Operations - the operations department pays the interest, e.g. more hardware required, incident analysis.
  - Development - the development team pays the interest, e.g. expensive user stories, low productivity, low developer satisfaction.
- Invest to Gain Insight - mentions whether additional investment is needed first to map the debt.
- Scope - describes the extent to which the debt affects the overall project (removed from the sanitized list).
- Comment - additional information (removed from the sanitized list).
- Data Availability - mentions whether there is additional information on the debt to be found in issue tracking software (removed from the sanitized list).

In addition to the aspects mentioned above, every item of TD was categorized based on the ontology, presented in Table 2.1. The type "Domain Level TD" was added to this ontology. The type "Environmental TD" was removed from the ontology, since it represents a supertype of infrastructure, social/people, process, build, and versioning debts. "Knowledge Distribution/Documentation TD" was renamed to "Documentation TD". Due to similarities and lack of data, "Testing/Test TD" and "Test Automation TD" were merged to "Test/Test Automation TD".

4. Results

This chapter provides an overview of the results of the data collection phase. The first section presents the results of the domain level TD case studies and the second section presents the results of the analysis of the list of TD items, collected from a project from the industry.

4.1. Domain Level Technical Debt in Practice

This section presents six case studies (CS), describing items of TD that can to some extent be related to domain level TD. The case studies have been conducted at QAware GmbH, a software company in Munich, during a period from December 2020 to February 2021. The structure of the case studies is described in subsection 3.2.1.

4.1.1. CS 1: Multi-tenancy

The Problem and its Context

In around 2015, two European telecommunication giants, Deutsche Telekom and Orange began jointly developing a voice processing platform. The purpose of the system is to pick up spoken audio of its users, talking to a smart speaker, to process the recordings by writing the voice to text, to analyze the intents, and finally to execute them. Voicification is a major trend that involves a number of popular technologies, such as Artificial Intelligence and Big Data, and is highly influenced by current privacy and security issues, like for example GDPR compliance in the European Union.

Even though the two companies are significantly late compared to their US counterparts, Telekom has an ambitious goal to conquer the market of systems becoming voice enabled. One of the main advantages of Telekom in achieving this objective is that they pay special attention to maintaining high privacy and security standards. Meanwhile, Amazon's recent privacy issues cast doubt on the GDPR compliance of their products. Therefore, Telekom has been able to profit from this gap and has the potential of growing even stronger, by offering well-targeted GDPR compliant products for the European markets.

Furthermore, Amazon is not concentrating on the services market and mainly targets private consumers with its smart speakers. Telekom, on the other hand, is primarily

targeting business users - their range of smart speaker products, besides securing their market position in many niches, is mainly the proving ground for their platform. Amazon might be reigning over the business-to-consumer (B2C) sector of voicification, but the race for business-to-business (B2B) has not even begun yet. There is a lot of potential in voice processing technologies and several areas that could benefit from it, for example hospitality, industrial, or health care applications. It is important to keep in mind that these areas generally have very strict requirements for privacy and security.

The project initially started as a cooperation between Orange and Deutsche Telekom and the goal was to share costs, so that it would become more affordable. Yet the dual client structure made management activities relatively complex and added some overhead to the massive project that involved over 300 people. A significant share of the manpower was provided by third parties, e.g. several software engineering companies participating in the development of the voice processing system. In addition to that, some of the work was also outsourced to Greece and India.

The presence of two clients, Deutsche Telekom and Orange, enforced the understanding that the system should have two tenants and thus a multi-tenant capability was introduced to the system. A multi-tenant software architecture means that multiple instances of a software system run on the same physical hardware but need strict separation of data. Multi-tenant hosting reduces the overhead of hardware management and makes it possible to share resources [42]. Hence, both Telekom and Orange would have their own tenant in the joint system and would share voice processing capabilities. At the time, though, no facility was conceived to allow a more fine grained split between different variants of each of the tenants, e.g. for different devices with different configurations, A/B-testing, canary releases, live testing and so on. When the need for such facilities became clear, the multi-tenant capability was misappropriated for this purpose. Technically, this worked to a limited extent, but the problems included the lack of lightweight deployment of new “tenants” and management facilities for large numbers of “tenants”. Actual tenants were not to be expected in such a number at such an early stage.

First issues with the initial one-product design emerged when it was discovered that language models would have to be continuously enhanced and this indicates that there would have to be multiple releases of products. All of these releases amount in multiple different types of configurations that are active at the same time. What initially seemed to be a good idea became invalid with time. The life cycle of the debt is illustrated in Figure 4.1.

From a domain perspective, though, a multi-tenant capability is something different and the correct solution should rather be a product-line of systems or simply a common platform that should serve as a base for configuring different products and services. There should be a way of instantiating and managing products and this was not designed to be a part of the system from scratch.

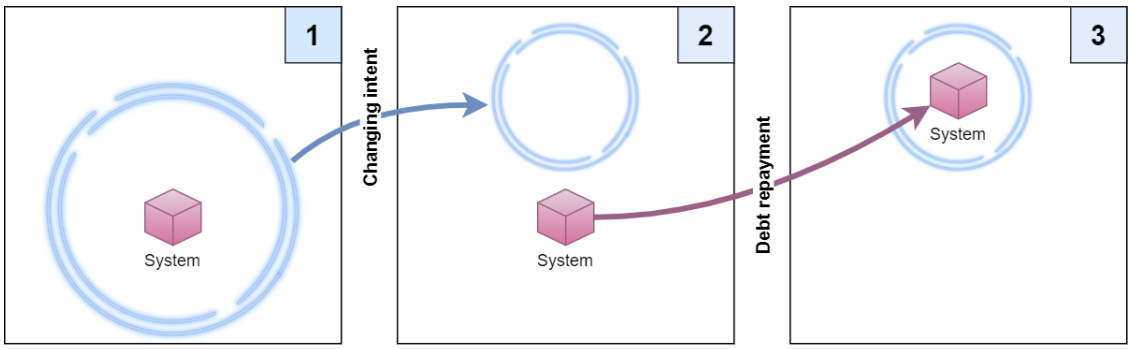

Figure 4.1.: The system was created with a lack of far-sighted design. The initial design with multi-tenancy support was sufficient at first, but as soon as the number of "tenants" started growing, the intent of the system changed and the domain drifted away. The debt arised due to the drift and brought an interest with it. Finally, the team started repaying the principal of the debt and the system was directed to its target domain.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated "high".

The debt is cross-cutting and affects both the system and the organization, including the management, processes, supporting systems, and legal aspects. It is considered contagious, as every new feature and change will be based on the incorrect design.

Principal of the Debt

There is no data to accurately quantify the principal of the debt. Yet the overall principal consists of three parts:

- Firstly, establishing a common, clear, and focused vision among stakeholders, so that everyone understands the domain of the system. This includes understanding the product landscape and the need for multi-configuration support.
- Secondly, involving the necessary experts and having sufficient technical system understanding to be able to implement the vision.
- Thirdly, implementing multi-configuration support in all parts of the system and the organization.

Interest of the Debt

There is no data to accurately quantify the interest of the debt. The interest of the debt consists of several limitations to the technological evolution and commercial success of the system.

- Lack of multiple configurations per tenant made it impossible to have frequent and cheap deployments of natural-language understanding (NLU) models, system components, or configuration changes for exploratory or test purposes. This in turn slowed down reacting to results from field trials etc.
- The lack of design and structure, supporting multiple configurations, made it difficult to manage larger numbers of versions and configurations.
- The system was not fulfilling its clients expectations, since the debt was slowing down progress and limiting the quality of the products.

Interest Payers

It is not clear who the exact payers of the interest were, but on the basis of the consequences the following can be concluded:

- The clients were paying a part of it due to their unfulfilled expectations and limitations to the commercial success of the project.
- The development team was paying a part of it due to the technological limitations, such as difficulties concerning the deployment and management of the different configurations.

Causes of the Debt

The prime mechanism that led to the debt is a changing intent. Use cases that were originally envisioned for the system changed and focused over time and the design of the system was not flexible enough to keep up with them. The situation was caused by a mixture of different causes:

- There was a lack of thorough initial planning and conceptual design.
- There was an absence of a common, clear, focused vision among stakeholders.
- There were deficiencies in the overall development process.
- There was possibly a conflict with agile methodologies that support rapid continuous learning.
- The complex structure, the large size of the project, and the existence of two clients made project management more difficult.
- The notions of multi-tenancy and multi-configuration support were complex and

novel to many people in the project.

Location in the TD Quadrant

The debt is considered inadvertent and prudent.

Management of the Debt

The debt has been noted and the team is to date still working on repaying the principal of the debt.

Relevance of Domain Knowledge

The debt is dependent on the domain of the system and domain knowledge was necessary for designing a solution and repaying the debt. It was difficult to initially imagine and envision the domain of the system, as the goal was to target a new market that had not been targeted by other competitors before. In addition to that, the clients were initially not completely sure what they expected from the system. The envisioned domain was vague and this led to the misplacement of the system. The domain debt is that the stakeholders did not understand that those were not tenants that they were developing, but product variants and initially valid decisions became invalid with time. In addition to that, the debt is not detectable directly from the source code.

Other Potential Types of TD

Having a design that supports multiple versions and configurations is located on the abstraction level of system design. Therefore, it can also be argued that it has characteristics of structural debt, for example architectural TD or design TD.

Since a possible cause is that people with required expertise and domain knowledge were missing from the development team, the debt can also be linked to social/people TD.

Observations

The following observations about domain level TD can be brought forward from this case study.

Firstly, the debt materializes only when also the domain of the system is taken into account. This means that the initial multi-tenant implementation of the system on its own is definitely not a bad design, but is simply being misused for something that it is not intended for. Domain knowledge plays an essential role in being able to understand

the debt and that it is caused by the actual domain drifting away from the envisioned domain.

Secondly, although the interest and principal of the debt cannot be quantified, they were both relatively high. The debt had spread throughout the system and the organization and its elimination required substantial redesign and rework that halted other important development activities. Due to limitations to commercial success, there was no alternative but to repay the debt.

Thirdly, the debt has characteristics of multiple types of TD. It is not exclusively domain level TD, but can also be considered as architectural TD, design TD, or social/people TD. However, it is important to take a look at the debt from a domain perspective, because then its life cycle becomes clear.

Fourthly, one of the causes leading to domain debt in this study is not investing enough time into designing and planning at the beginning of the project. If the design phase would have been longer and more thorough, it is possible that the envisioned domain would have become more focused, making it possible to avoid the debt. A longer design phase at the beginning of the project would have probably cost less than having to deal with the debt in later stages. The structure and flow of the project were based on SAFe and Scrum and a long initial planning phase to some extent conflicts with these agile practices. This suggests that a compromise should be found between agility and detailed planning with regards to avoiding domain debt.

4.1.2. CS 2: Database Problem

The Problem and its Context

This item of debt is taken from the list of TD items, described in section 4.2. Therefore, the context of the project, described in subsection 4.2.1, applies for the item of TD in question. The debt was discovered at the beginning of the transition in an analysis phase. The life cycle of the debt is illustrated in Figure 4.2.

The data-centric system is used to make inquiries and order car parts for a given vehicle, e.g. in the context of a repair. Having prior domain knowledge about cars makes evident the fact that the system and its data model have to be able to cope with a relatively large amount of data, as a modern average car typically consists of around 30.000 individual components [43]. Various car models can have arbitrary combinations of parts and therefore they have to be aggregated and joined in different ways. Understanding the domain is therefore important to be able to design a sufficiently efficient data model that is capable of executing requests in reasonable time.

An initial code and data review revealed that the existing data model had strong deficits in efficiency. The system was utilizing a standardized normalized entity-relationship data model that is optimized for editorial use cases, i.e. commissioning

and compiling data. This model was partially also being enforced by the selected technologies: an Oracle database and object mapping technology. Yet the main use case for the system is being able to research and find data, for instance, to determine whether a part is valid for a given vehicle. Doing this with the editorial data model is rather inefficient, as the data is distributed over multiple tables and has to be joined at runtime.

Considering that the system is estimated to have over 50.000 users worldwide, the debt has a strong negative influence on performance and leads to frequent system outages at peak times. In addition to that, the existing data model has not been documented by the previous team and this led to maintenance issues, as it was difficult to find defects or extend functionality without understanding the underlying model.

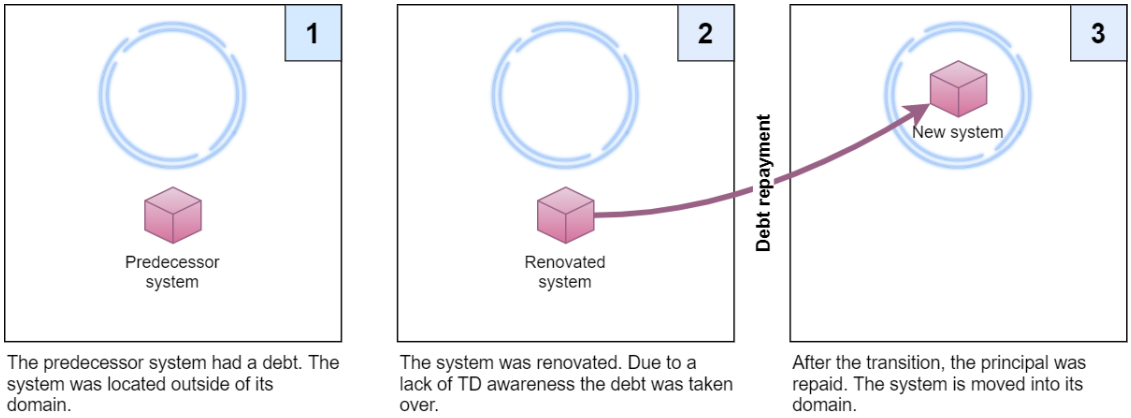

Figure 4.2.: The debt was already present in the first version of the system. It had a data model that was not capable of efficiently fulfilling use cases from its domain. The old system was renovated in 2015, but due to a lack of TD awareness, large parts of the old system were simply copied and with it also the debt. In 2020 there was a transition and the new team identified the debt and started repaying its principal.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated “high”.

The debt had spread all over the database and the application and it was relatively contagious, as new data and features were advancing the problem even further.

Principal of the Debt

The principal of the debt consists of two main parts:

- Firstly, short-term firefighting: quick-fixes were done to keep the system in operation. This involved analyzing and optimizing a smaller set of selected queries, e.g. introducing bulk queries, adding missing indices to the database, upgrading the database infrastructure.
- Secondly, since the system is meant to undergo a cloud transformation, the complete data model would be redesigned as a part of it. As a preparation to that, the business data model and particularly its validity rules would have to be documented.

An estimate to the principal can be given based on data collected from issue tracking software. Currently the data only describes the first line of action, as the cloud transformation is still on its way. There are three XL-sized stories, one L-sized, one S-sized, and one XS-sized story that deal with finding and optimizing calls and introducing database indices (see Table 3.1 for information about task sizes).

Interest of the Debt

The interest of the debt manifests itself as limitations to the performance of the system, which entails costs for both the organization and the users.

- Low system performance and frequent system outages. The system would often be completely unusable during peak hours.
- The budget of the operations department would have to be increased continuously, as they have to order more and larger web servers to be able to increase the database performance.
- Outages lead to reports that have to be analyzed and thereupon, to quick-fixes that have to be designed, implemented, and deployed for increasing the performance and stability. For instance, the task to analyze database problems and to specify various performance monitoring checks cost the project in total an effort of the size XL in the spring of 2020 (see Table 3.1 for information about task sizes).
- Outages also lead to various systemic processes, such as the overhead of service interruption reporting.

Data collected from issue tracking software gives further insight into the scope of this interest. There are multiple incident reports (INC) and change requests (CR) related to low database performance and outages.

- A sprint in the summer of 2020 had three unplanned service interruptions, that led to one INC and one CR.
- A sprint in the autumn of 2020 had two unplanned service interruptions, that in total led to four INC-s and one CR.

Interest Payers

The interest on the debt is paid first by the users of the system and from then on the cost extends to units within the organization.

- The users of the system suffer from low system performance and outages. Car dealers and workshop employees are not able to do their work during peak hours and this can be directly mapped to costs.
- The business department owning the system would pay the interest, as they have to counter system outages and spend a significant amount of the budget on maintenance and operations tasks.
- The development team would have to analyze reports and work on quick-fixes. This would take away time and budget from other core activities, like implementing new features. Additionally, missing data model documentation makes maintenance and feature development more time-consuming.

Causes of the Debt

The main mechanism that led to the debt is a lack of TD awareness. The debt was introduced into the system at its inception, during initial design phases. Due to the transition and the lack of documentation there is no information available from that time that would shed light on the exact causes leading to it. The new team at QAware GmbH discussed the matter with members from the previous team and it turned out that they were not aware of the consequences of the editorial design and did not even contemplate redesigning the data model. The following is a list of potential causes:

- There was a lack of initial design and planning.
- There was a lack of domain knowledge. The actual use cases might not have been known to the previous teams.
- There were business trade-offs, i.e. not having enough time or money to invest into designing an appropriate data model.
- There was convenience. The core of the data model was undocumented and the code was basically copied from the predecessor system, needing to be renovated. Simply copying the old editorial data model from the predecessor system was much cheaper and quicker than investing time into redesigning it.
- There was a lack of expertise in the team. None of the former team members might have had the competence to detect early that the editorial model will not be able to satisfy the use cases of the system.

Location in the TD Quadrant

The debt is considered inadvertent. Depending on the causes it could be either reckless or prudent.

Management of the Debt

The decision was to repay the debt and the general goal was to restructure the data model so that it reflects the correct use cases. To date the team has applied multiple quick-fixes, has completed the documentation for the new model and has started the cloud transformation with a first version of the optimized data model.

Relevance of Domain Knowledge

Although the debt is apparent from performance metrics, domain knowledge is needed for formulating appropriate removal strategies. Some aspects of the debt can be detected and removed without domain knowledge: for example, bulk requests to fetch texts for a webpage instead of individual requests for each text item. However, the core of the problem is that the information needed for a use case is spread over many tables and was not documented. Improving that part of the debt is only possible with domain knowledge. Relevant domain knowledge includes knowing the actual use cases, but also the semantics of the data model.

Other Potential Types of TD

An inappropriate data model lies on the abstraction level of system design. Thus, the debt can also be linked to structural types of TD, such as design TD or architectural TD.

A possible cause of the debt is that there was a lack of competence in the previous teams. Team related deficiencies can be linked to social/people TD.

Observations

A number of characteristics of domain level TD can be derived from this case study.

Firstly, the debt does not necessarily mean that the implementation of the system is wrong, as long as the use cases are not taken into account. An editorial data model could be an optimal solution in certain cases. The debt materializes and becomes visible only if also the domain is taken into consideration. Meanwhile, other types of TD, such as certain antipatterns or bad code practices, can remain a debt even without knowledge of the domain.

Secondly, in this study the debt does not automatically affect the developers first. The main interest payers are the users of the system. If the system would be developed at

a pre-live stage without any active users, the developers would probably not notice significant consequences, i.e. development activities would not be hindered. Only when the system is up and running and the use cases take effect, will the development team become aware of the performance issues.

Thirdly, both the interest as well as the principal are noticeably high for this case. The interest keeps rising as the number of users and the amount of data increases. Not only the interest, but also the principal has a growing trend. When new functionality is added on top of the incorrect data model, the amount of subsequent rework also grows. The debt spreads throughout the core of the system and its elimination can only be achieved by a redesign.

Fourthly, it is difficult to argue that the debt exclusively belongs to one certain type of TD. It has characteristics, which belong to the categories of several different types of debt. For instance, it could be considered design TD, architectural TD, social/people TD, or domain level TD, depending on the perspective.

Fifthly, all activities related to the debt need a certain amount of domain knowledge. Not only avoiding the debt at the beginning of the renovation process would have required domain understanding, but also being able to fully understand and eliminate the debt later on.

4.1.3. CS 3: Interactive Filtering

The Problem and its Context

The following case study describes a situation, where an agile project is struck by hidden requirements that arise and conflict with the design and capabilities of the system. The system in question is an analysis tool developed for a German telecommunications company that allows them to analyze the usage of their voice-guided digital assistant, i.e. smart speaker. Work on the tool began in the spring of 2019 and it is basically a database that contains speech processing data and allows its users to query information about usage patterns. The system lies in the domain of voicification and the team developing it should therefore be aware of the concepts behind voice processing, the structure of the processed data, and how the data is used.

Due to the agile and exploratory nature of the project, the team did not invest a lot of time into planning and simply used what was available and well known to them, an SQL-based solution, based on PostgreSQL. The team initially provided a very small set of selectors for the data, effectively just the date period and three flags.

After the first few deployments, though, it became obvious that using the tool to interactively filter and browse through the data would be a valuable asset. Hence, soon the SQL-based solution came to its limits, when 13 additional selectors had to be added to the existing date and flags. Worse, some of these selectors had to support both exact

and approximate matching, i.e. matching patterns or substrings, and this is relatively difficult to achieve with a classic SQL-database.

The debt lies in the fact that the team failed to correctly weigh between a traditional SQL-database, targeted at executing transactions, and a column store or indexing database, which is better suited for data analytics. Traditional SQL-databases will fail to support data analytics, as they are not capable of performing statistical aggregation in reasonable time [44]. A column store database stores data records in columns rather than in rows and is reasonably efficient in executing statistical computations [44].

Despite not fulfilling some user expectations, the tool was a big success, as marked by the rapidly increasing user basis, which put even more load on the system. Simultaneously, the amount of data grew as well, further stretching available resources. All in all, although the system was implemented with high quality from a technical point of view, users had needs beyond the initial design envelope. The team failed to envision the growth of data and the evolution of usage scenarios. The life cycle of the debt is illustrated in Figure 4.3.

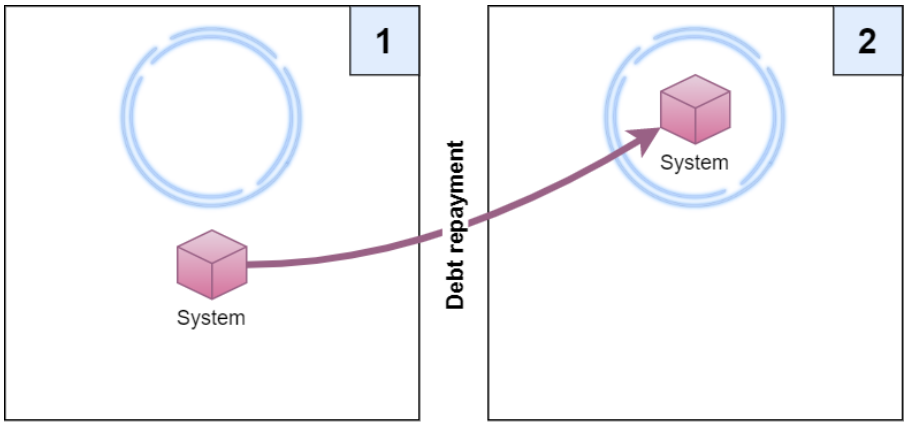

Figure 4.3.: The system was created with a lack of far-sighted design and planning. The team decided to rely on familiar technologies that turned out to be unsuitable for addressing the actual domain. The debt became visible through user experience. The team decided to repay the principal of the debt and began redesigning the system.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated “high”.

The debt had spread all over the architecture and database of the system. It was contagious, since it was present in the foundation and was affecting new features. Growth of data was also spreading the debt.

Principal of the Debt

There is no data to accurately quantify the principal of the debt, but it is estimated that it was approximately a man-year of work. The repayment of the principal includes two steps:

- The first step was to use the indexing capability, offered by the cloud computing service Azure, to slightly improve the performance of the tool.
- The second step was to redesign the database and to gradually move away from the SQL-based design.

Interest of the Debt

There is no data to accurately quantify the interest of the debt. Yet it imposed limitations on the expected usage of the tool and prevented growth into potential application areas.

- The performance of the system fell with the growth of data and amount of requests.
- Many features had to be delayed in order not to add too much load on top of the inadequate database design and technology.
- Failure to resolve the issue in a timely manner would prevent the system from getting the reach and traction necessary for the overall project.

Interest Payers

The team understood that the technological basis would not be viable beyond the first few iterations of the project. The interest of the debt was paid by both the development team and stakeholders outside of the team.

- The users of the system were affected by the low performance. They were not able to explore data the way they expected to.
- Due to technological limitations, the development team could not further develop the system. They were aware of the performance issues and were receiving feature requests from the users.

Causes of the Debt

The debt is mainly caused by hidden requirements, which is the inability to foresee the rapid growth of the data volume and the shift in usage patterns. A combination of the following causes led to the debt:

- There was a lack of initial design and planning.
- The project was following agile principles. Agile tells you to start simple, do only what is immediately ahead of you, and not to look any further into the future. The idea behind this standpoint is to continuously learn as you go and not to end up in an analysis paralysis at the beginning of the project. Yet some detours, such as the one in this study, are simply too expensive and should be avoided. Core design decisions, architectural and technical choices form the foundation of a system and are very difficult to alter later on.
- The development team decided to utilize familiar technologies. This can be compared to the Golden Hammer antipattern. It is described as a situation, where developers tend to blindly rely on technologies that they are familiar with [45]. Simply because one knows how to use a hammer well, does not mean that it is the best tool for turning a screw into the wall. Similarly in this study, the team simply assumed that PostgreSQL would do well and started with a known and reliable technology.

Location in the TD Quadrant

The debt is considered inadvertent and prudent.

Management of the Debt

As soon as the debt was identified, the team started designing and implementing a new solution. It is highly likely that the transactional core of the system will remain and that the new solution will simply add a layer on top of it. To date the team is still working on repaying the debt, but a large degree of it has already been resolved.

Relevance of Domain Knowledge

Understanding the domain of the system and specifically its use cases, plays an important role in recognizing that the SQL-based database solution is insufficient. The debt emerged due to a misjudgment of the value proposition for users and could have potentially been prevented with a more customer-centric design. It became evident once the system actively entered its domain, i.e. the users started using it. Domain knowledge played a role in designing and implementing a new and more performant solution.

Other Potential Types of TD

The debt could belong to the category of design TD or architectural TD, since an incorrect database model lives on the abstraction level of system design.

The Golden Hammer antipattern can be linked to social/people TD. There was potentially a lack of team members, who would be experienced with different database technologies.

Observations

The following observations can be made about the characteristics of domain level TD from this case study.

Firstly, domain knowledge plays a significant role in the emergence of the debt and later in repaying it. The relevant domain understanding in this case consists of being able to envision the actual use cases and secondly realizing how the domain shifts and develops in the future. The debt itself only exists in combination with the domain, as there is nothing wrong with the way the system has been technologically implemented.

Secondly, the debt is caused by a lack of initial design and planning. This study highlights a conflict between agile methodologies, where it is advised to learn as you go, and domain debt, the avoidance of which requires thorough initial analysis.

Thirdly, both the interest and principal of the debt are relatively high and first affect the users of the tool. The development team is not directly affected, as the debt does not necessarily slow down tasks related to the implementation of the system. Indicators of the debt are low performance and feedback from users.

Fourthly, the debt is a combination of multiple types of TD. As discussed earlier, it can be seen as domain level TD, architectural TD, design TD, or social/people TD. All of these types are important, as they concentrate on different perspectives. Domain level TD highlights the causes and life cycle of the debt, architectural and design TD highlight the location and nature of the debt, and social/people TD points out to a potential cause. So it makes sense to use it in combination here.

4.1.4. CS 4: GDPR

The Problem and its Context

A further example of domain level TD comes from the variegated landscape of voice-based digital assistants (VBDA), such as smart voice hubs and speakers. Such devices have recently raised some privacy concerns, that have been fueled by cases, where conversations of users, recorded by these devices, have leaked and been misused.

Amazon started developing its smart speaker called Amazon Echo that incorporates a virtual assistant called Alexa already in 2011 and released it to the public in the autumn of 2014 [46]. The speaker turned out to be a major success and Amazon was able to sell three million of these devices up to 2016 and the consumer feedback was also very positive [46]. Four years after the release of Echo, the European Union (EU) enforced a new data protection law, called the General Data Protection Regulation (GDPR) [47].

With the GDPR in effect, the focus of privacy discussions also fell on smart speaker devices that were sitting in the homes of users, listening to conversations. These discussions became especially intense, after Amazon accidentally sent 1.700 Alexa voice recordings to the wrong user following a GDPR data request [48]. The user from Germany had requested his data from Amazon and to his surprise also received a bunch of audio files, even though he did not own and never had used an Alexa device before. It turned out that Amazon had made a mistake and accidentally included the data from someone else. And as if this was not enough, Bloomberg reported a year later in 2019 that Amazon workers were listening to recordings of users talking to Alexa [49]. Amazon claimed that they were reviewing some recordings in order to improve their voice recognition technology [50], but the users were not happy with it. Amazon's failure to meet privacy regulations can be seen as domain debt. The life cycle of the debt is illustrated in Figure 4.4.

The smart voice hub developed by Deutsche Telekom, called Magenta, took a different approach and paid a significant amount of attention to privacy already since the beginning of the project. The company had already been burnt by an eavesdropping scandal in 2008 [51] and took privacy-related matters more seriously this time. Magenta, released to the German market in 2018, was hopelessly late compared to Amazon's Echo, but has the advantage of implementing strict German data protection laws and the GDPR since its inception [52]. Based on experience, it is much easier to implement privacy protection measures, when they are already considered in the initial design of the system, claimed to be the case with Magenta, than to try to implement them later after years of development, as seems to be the case with Amazon Echo.

It is important to note that these impressions and observations have been derived, as information about internal privacy measures of Magenta and Amazon Echo are not available to the public.

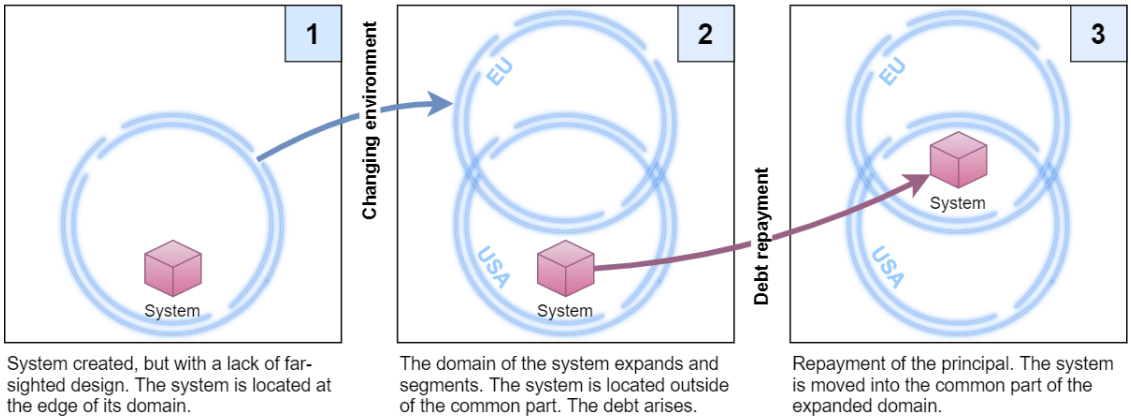

Figure 4.4.: The system was created without far-sighted planning and design. The team did not pay enough attention to privacy and failed to foresee cultural and regulatory developments. Soon the environment changed, GDPR came into effect in the EU, and the domain expanded and segmented. The system was failing to address the more strict domain in the EU and the debt arose. Both technical and organizational changes are needed to repay the principal of the debt. In addition to that, it requires regaining trust in the market.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated “high”.

It is contagious and spreads all over the structure of the system, internal and external processes, supporting systems, and tools, down to the organization and its cultural mindset.

Principal of the Debt

The principal of the debt is large and probably amounts to many years of work on redesigning the system, reforming the organization, and reestablishing trust in the marketplace. It consists of the following two main parts:

- New features have to be added, such as informed consent, privacy by default, information on data usage, security, deletion of data etc.
- Developers and employees, who work with data, have to be trained accordingly and certain security measures have to be implemented, such as stricter access control, inverse transparency, and data usage protocols. This requires a cultural shift and might even need a change of staff.

Interest of the Debt

It is impossible to accurately quantify the interest of the debt, as internal data from companies developing VBDA-s, such as Amazon, is not public. However, it is composed of two main parts that limit the commercial success of the product.

- Firstly, VBDA-s that do not pay enough attention to privacy regulations will fail to address use cases from their domain. Especially use cases that require high privacy and security standards.
- Secondly, breaking regulations will ultimately lead to fines. For example, violations of the GDPR are increasingly being prosecuted and fines amount to 4% of a perpetrators annual worldwide turnover.

Interest Payers

The debt does not necessarily affect the developers, by for instance slowing down the development of the system. It is rather independent of the technology. The interest is mainly paid at a higher level:

- The marketing and sales departments pay the interest, since they are not able to sell the product in certain markets. For example, non-GDPR-compliant products cannot be sold in the EU.
- The users pay the interest, since their privacy requirements are not fulfilled.
- The CEO of a company is generally held responsible for privacy related topics and could ultimately be ousted by board members, if the debt is not repaid in time.

Causes of the Debt

A changing environment is the central cause behind this debt. The general society is becoming more and more aware of the importance of the privacy of their data and with it, various supporting regulations are launched and improved. In this context, two potential causes for the debt are:

- Cultural blindness and failure to understand and map cultural differences. Most VBDA-s, such as Amazon Echo, were developed in the United States, where privacy has a much lesser meaning than for instance in the EU. It is possible that the development team overgeneralized the domain of the system based on their own social and cultural bubble, disregarding other settings.
- Business trade-offs. It might be that the development team was completely aware of taking on the debt and privacy was considered at some point during the design phase, but it fell short of other goals, such as gathering massive amounts of data, in order to improve their deep learning algorithms.

Location in the TD Quadrant

The debt is reckless and could be both inadvertent and deliberate, depending on the cause.

Management of the Debt

There is no data available on how companies, such as Amazon, have managed the debt.

Relevance of Domain Knowledge

The fact that Amazon Echo has a deteriorated privacy reputation, can be identified as domain debt. There are several use cases in the domain of smart speakers that require high privacy and data protection measures, like for instance medical applications. Magenta will probably be more successful in such use cases than Amazon Echo. Even though Amazon Echo might fulfill high software quality standards and have a larger market share than Magenta, it has issues with its privacy-related domain design. Regulatory compliance is totally disconnected from technology.

Other Potential Types of TD

A possible cause of the debt is that the development team was not aware of the cultural and regulatory differences between different markets and countries. This absence of expertise and knowledge in the team can be linked to social/people TD.

Observations

This case study allows the following observations to be made in regard to domain level TD.

Firstly, the debt is characterized by a very high interest rate and principal. The interest would ultimately lead to the product being banned in certain markets and to a considerable reduction in the number of users. As for the principal, it is not even clear whether the organization will be able to repay it, as it requires both technical as well as major organizational and cultural changes. Even if the organization is able to change both the system and the way they work, regaining trust among users is even more difficult. An example of this is the case of Huawei and their 5G technology, which is said to be an instrument of the Chinese state that they can use to influence Western countries. Let us hypothetically picture that Huawei tells the truth and their technology and organization are secure and independent of China. Regardless of their continuous efforts and marketing campaigns, their reputation is permanently damaged and many people will still have doubts about the security of their products and their compliance

with privacy requirements. Changing this would require a generational change, a change in Chinese policy, or selling and moving the whole company to a more reliable country.

Secondly, in this case the cause of the debt could be both deliberate and inadvertent. It is possible that the development team was aware of the privacy requirements, but they decided to take the risk and ignore them due to their rather restrictive nature, or they simply made a miscalculation and did not pay enough attention to the cultural discrepancies in the different markets.

Thirdly, the debt is independent of the technology. No static or dynamic analysis tool will be able to detect it and it requires a certain degree of domain understanding to discover the issue. The problem does not result from the system's implementation itself, but from the conflict between the system and its environment, specifically unsatisfied requirements and regulations.

4.1.5. CS 5: Ground Truth

The Problem and its Context

The problem in this case study is independent of a technical system and lies within a socio-technical context, including the organization and individual stakeholders. The system in question is a machine-learning based digital speech assistant.

The issue revolves around the Ground Truth (GT) of the system, which is basically a set of labelled data that is used as a reality check for machine learning algorithms [53]. In the context of a speech assistant, this data consists of different utterances. The GT and the labelled utterances serve as a basis for the Natural Language Understanding (NLU) component of the speech assistant. The GT determines how well the application can interpret audio commands of its users and is therefore an important part of the system, which is associated with many different tasks and responsibilities.

Due to the cross-cutting nature of this component, there is no single clear responsible unit or person for it, which means that it is used by many different parties, but none of them are fully responsible for its quality, integrity, and availability. This organizational uncertainty puts the system at risk.

There are four different types of stakeholders, who are involved with the GT and are potential candidates to take responsibility for it.

- Firstly, the team labelling the utterances based on a predefined set of rules, which mainly consists of low-wage auxiliary workers, for instance students.
- Secondly, a team of software engineers and tool builders, who are responsible for implementing, maintaining, and operating the database that holds the GT. However, they lack the linguistic insights that are needed for understanding the content of the GT and are therefore unsuitable for being responsible for it.

- Thirdly, there are several feature teams that are responsible for certain functional aspects of the speech assistant. Therefore, they could be seen as responsible for those individual parts of the GT that concern their respective features.
- Fourthly, a team of computational linguists, generally with little engineering skills, but who are capable of understanding the content of the GT. The team of linguists would be the best candidate for taking over the responsibility of the GT, but this is not the case.

The debt is therefore that the lack of organization of processes and data has led to the GT growing into a monolithic chunk of data, reaching 86 MB. The rather large size and unorganized design makes many tasks involving the GT rather difficult and risky. The quality of the data also suffers, as there are no clear data structures and people are not able to fully understand its content. Considering that this data is being fed into the training algorithm of the voice assistant and should ultimately result in a well working engine, it can be considered highly critical. The life cycle of the debt is illustrated in Figure 4.5.

Domain knowledge played an important role in identifying and understanding the debt. It was discovered, when the team at QAware GmbH was tasked with learning and analyzing some activities and processes in the immediate proximity of the GT.

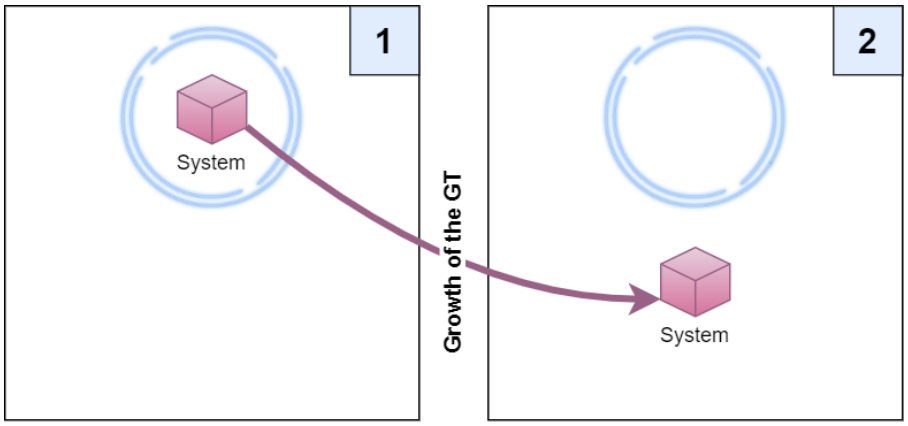

Figure 4.5.: At the beginning, when the system was created, the Ground Truth (GT) was rather small and did not cause any difficulties. As it started growing, it became more and more difficult to have an overview of its content. None of the stakeholders worked on maintaining its quality and a debt arose. With the debt, the system shifted out of its domain.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated “high”.

The interest of the debt keeps rising due to its highly contagious nature, resulting from the fact that the GT is a central part of the system that affects various other components. The debt has spread throughout the system and the organization.

Principal of the Debt

The principal consists of organizational and technical changes:

- From the organizational side, responsibility for the GT would have to be assigned to the correct stakeholder, who would be taking care of maintaining its quality, integrity, and availability. The main obstacle is making sure that all parties understand the seriousness of the problem and thereupon trying to solve the organizational blockade.
- From the technical side, it is estimated to require around 20 man-days of work that involves splitting the GT into intelligible data segments, which could be handled more easily, and putting them under proper version control.

Interest of the Debt

The interest imposes limitations on both technical and organizational aspects of the project. Furthermore, it limits the commercial success of the product.

- The lack of a proper data structure hinders backing up, versioning, improving, and extending the GT.
- It obstructs the deployment of NLU models, distributed working, and the development of new features.
- The monolithic nature of the GT also leads to high maintenance costs and will consequently reduce the accuracy of the smart speaker, as its ability to properly interpret voice commands directly depends on the GT.
- It will cause loss of customer trust and esteem and will ultimately endanger the success of the complete marketing proposal.

Interest Payers

The interest is paid by both internal and external stakeholders:

- The users of the smart voice assistant would experience a relatively high error rate, rendering the device impractical. It would fail to effectively fulfill their use cases.

- The sales and marketing departments would no longer be able to sell the product successfully and this would result in financial loss to the company.
- Developers are not able to deploy new skills and are slowed down by misunderstandings, caused by poor data models.

Causes of the Debt

The central causes behind the debt are disorganization, a lack of management, and a lack of initiative.

- The organizational processes and responsibilities concerning the GT are either vague or even non-existent.
- The management is aware of the situation, but refuses to take action and argues that the teams below need to organize themselves. Yet it is difficult for lower parties to effectively take up any cross-team initiatives, considering that the number of people who have played a role in the project is reaching 500.
- Some of the involved parties have a lack of willingness to deal with the issue and it is possible that some individuals benefit from it, by keeping certain knowledge only for themselves and thereby trying to increase their importance in the project.
- The team of linguists could have noticed this problem at an early stage, understood that it was their area of competence and could have resolved it quickly. However, they do not have the skills and willingness to technically approach the problem.

Location in the TD Quadrant

The debt is considered reckless and could be both inadvertent and deliberate.

Management of the Debt

To date, the debt still exists and it has not yet been properly managed. The side of the team, represented by QAware GmbH, has identified and mapped the issue and is trying to draw the attention of others to it, so that work on a solution could be started.

Relevance of Domain Knowledge

The problem is not dependent on technology, as it is rather easily possible to handle 86 MB-s of data from a technological perspective. But for a human it is somewhat difficult to understand and use the whole data set as a large unstructured chunk. In addition, resolving the debt is not a major challenge from a technical point of view, but it requires a significant amount of organizational effort. Domain knowledge is needed to understand the scope of the interest of the debt.

Other Potential Types of TD

The issue can also be linked to social/people TD, which covers inefficient social structures and organizational forms, and even to versioning TD, considering that the GT is not under version control.

Observations

The following observations with respect to domain debt can be drawn from this case study.

Firstly, it is caused by a lack of vision and ownership. Since none of the stakeholders feel responsible for the GT, no one feels obligated to take care of its quality. The upper management, who should coordinate the solution of the problem, does not seem to understand its extent.

Secondly, domain knowledge is needed to perceive the scope of the interest of the debt. The debt is not a major problem from a technical perspective, but rather from a domain-oriented organizational perspective.

Thirdly, the interest and principal of the debt are relatively high, because they cross the boundaries of the system and also affect the organization. Several parties, including the users, management, business departments, and development teams, have to pay interest on the debt and ultimately have to work together in order to be able to repay the principal.

Fourthly, the debt also has features of social/people TD and versioning TD. They are valid classifications, but in isolation, they are not able to characterize the full extent of the debt. Social/people TD is limited to organizational aspects and versioning TD to technical aspects. So it seems sensible to use them all together.

4.1.6. CS 6: Crawler Attacks

The Problem and its Context

This item of debt is taken from the list of TD items, described in section 4.2. Therefore, the context of the project, described in subsection 4.2.1, applies for the item of TD in question.

The system was first renovated by a previous team and was then handed over to a team at QAware GmbH. Most of the renovation was done without further planning and large parts of the old system were simply copied. Along with this, the renovated system also brought with it several shortcomings of the old system.

The application offers a client/browser user interface that is serviced by a number

of REST endpoints. The problem is that security requirements were not prioritized for those endpoints and they have been implemented in a straightforward manner, which makes them rather easy targets for crawling attacks. They simply require authentication and then send the response data in plain JSON. There are no safeguards, nor monitoring mechanisms that would make it possible to detect crawling attacks. At the same time, the data is very important and valuable to a part of the stakeholders of the application and should be properly protected.

The problem is not only theoretical, as the new development team has already noticed and observed crawling attacks against the system. The debt was identified during an architecture analysis phase at the time of the take-over from the previous team. The issue was relatively obvious, considering the amount of REST endpoints and the absence of safeguards and monitoring measures. It means that no special domain knowledge was required in order to be able to detect and map the problem. The life cycle of the debt is illustrated in Figure 4.6.

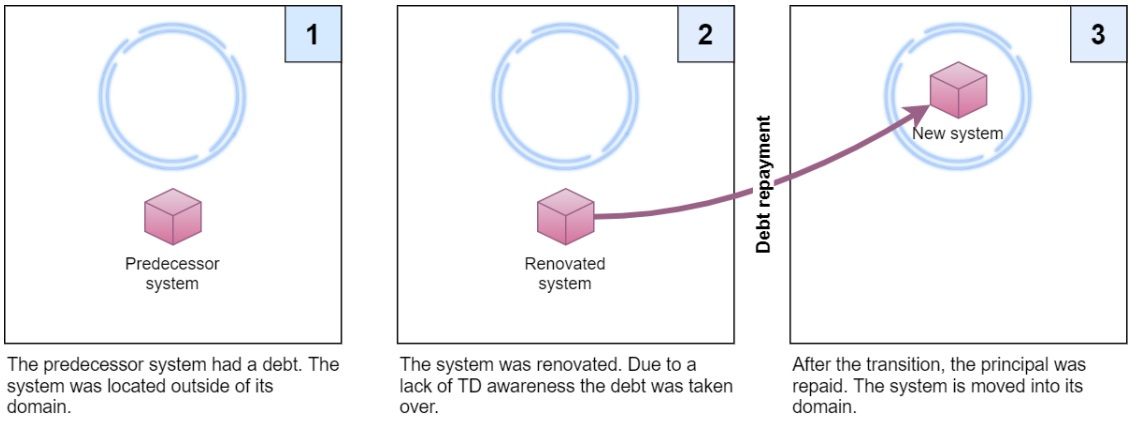

Figure 4.6.: When the system was created, it most probably did not have any measures to protect its endpoints against crawling attacks. Since it was installed directly at the customer's site, the debt did not become evident. The system was later renovated with a lack of TD awareness and the renovating team did not identify, nor repay the debt. After the transition, the new team identified the debt and started repaying it due to its high interest.

Classification and Observation Data

Criticality and Properties of the Debt

The criticality of the debt was rated "high".

At the time of the discovery, the problem had already spread throughout the architecture and design of the system, affecting all of the endpoints. The debt is not considered to be contagious.

Principal of the Debt

The principal of the debt consists of the following tasks:

- Implementing monitoring and alerting mechanisms to detect crawling attacks.
- Automatically locking or deactivating user accounts. Suspicious users are added to a blacklist and the application will refuse to service them.
- Reworking the architecture and design of the system, so that it would include additional safeguards, such as having a speed throttle to slow down users who exceed a certain threshold of requests per time.

Monitoring and alerting mechanisms have already been implemented and data collected from issue tracking software gives some insight on how expensive this part of the principal was. In total three tasks with sizes M, L, and XL are related to the implementation of these mechanisms. Work on the automatic lock-out of user accounts is not yet complete at the time of writing this case study, but it has been estimated to be a 5-10 XL task (see Table 3.1 for information about task sizes).

Interest of the Debt

The crawling attacks have caused loads on the system and the team has had to work on maintaining the performance. The interest consists of the following parts:

- Crawling reduces the system performance.
- The attacks have ultimately led to system outages, causing the overhead of having to detect and analyze their causes and restoring the crashed services.
- The analysis and resolution of incident reports, errors, and monitoring alarms from log files that have been triggered by defective or erroneous REST calls.
- Log files have to be secured in order to enforce non-repudiability and to have legal proof of crawling.
- The accounts of suspicious users have to be deactivated manually, which is estimated to be around 30 minutes per user account.

As of May 2020 there is also data from issue tracking software that can be used to put the interest rate of this debt into numbers. The first crawling attacks were identified in the beginning of May, when there were a lot of failed requests with error codes. The deviations and errors were investigated using visualizations and data from logs. A specific user was identified, targeting a database. As a result, the user was blocked. Since then, the team has had to deal with a total of at least 51 cases, where different user accounts have initiated crawling attacks against the system. Considering that blocking one user takes about half an hour, the business department has invested over 25 days to address this issue in a period from May 2020 to December 2020. This effort does not include identifying the user, saving the logs, or analyzing the targeted endpoints.

Interest Payers

The interest is paid by both internal and external stakeholders:

- The low performance and recurring outages are mainly affecting the users of the system, who are either requiring more time for their tasks or are completely unable to do their work.
- The operations department has the additional cost of improving and adding hardware, to increase the performance of the system. Besides that, they have to invest time into investigating incidents and restoring crashed services.
- The product owner and the business department on the client side have to deal with manually deactivating accounts.
- Developers must implement crawling alerts and have the additional effort of securing log files of crawlers, whose accounts have been disabled.

Causes of the Debt

The main causes of the debt are assumed to be a combination of:

- missing knowledge,
- business trade-offs, and
- a lack of initial planning and design.

It is likely that the previous system also did not have any security mechanisms and guards against crawling attacks and were therefore not considered relevant for the newly renovated system. At the same time, crawling is relatively common in practice and it is a broadly accepted standard to protect endpoints against such attacks. It is therefore unclear why this was not taken into account when the system was renovated.

Location in the TD Quadrant

Unknown, most likely inadvertent and reckless.

Management of the Debt

Due to frequent attacks, resulting in manual labor, the new team at QAware GmbH decided to partially pay back the debt. This includes implementing monitoring and alerting mechanisms and the automatic deactivation of suspicious users. Full repayment has not been considered yet.

Relevance of Domain Knowledge

Features speaking against domain debt are that in general, domain knowledge is not required to detect and solve this problem. Crawling is a relatively widely spread practice and most software developers should be aware of the issue, especially, if the application offers public endpoints. The new team at QAware GmbH did not need domain knowledge to identify the debt.

From the perspective of requirements, it is possible to make systems secure against crawling attacks and there exist practices that make crawling difficult. The fact that these requirements were not formulated or prioritized during the renovation process, can be linked to a lack of insight into the domain and/or missing experience.

The predecessor system was a decentralized application and the data and clients were installed directly at the customer's site. Therefore, it was impossible to notice any crawling activities back then. However, in the more centralized renovated application, which is used by a high number of resellers, this is not the case anymore.

Other Potential Types of TD

Secure implementation of endpoints lies on the abstraction level of system design and can be linked to structural debt, such as architectural TD or design TD.

Links to social/people TD can also be observed, as it is possible that there was a lack of experienced developers in the previous team that renovated the system.

Observations

The following observations about domain level TD can be made from this case study.

Firstly, the debt has a relatively high interest rate and criticality, mainly caused by the frequent attacks that each require considerable amounts of attention. Yet in comparison to previous case studies, the principal of the debt is located in the lower-end area of features. Repaying the debt does not necessarily require rebuilding large parts of the system or making major organizational changes.

Secondly, the preeminent indicators of the debt are low system performance and feedback from users. With a certain degree of experience and knowledge about crawling attacks, it can also be detected directly from implementation level artifacts, such as the source code.

Thirdly, the debt might have been caused by missing domain knowledge, which in turn resulted in an inadequate set of requirements for the system. But overall, it does not play a major role later on in detecting and repaying the debt. Because of this, the debt lies on the borders of architectural/design TD and domain level TD.

4.2. Proportions and Impacts of Technical Debt in the Candidate Project

The list of TD items, collected from the Candidate Project, described in subsection 4.2.1, contains a total of 87 items. Due to confidentiality requirements, it is not possible to publish the original list, but a sanitized version of it is available in [54]. The structure of the list is explained in subsection 3.3.1. Below we present the results from this list.

4.2.1. The Candidate Project

The Candidate Project utilizes agile methodologies, employs a well-established set of DevOps practices, and has a product-oriented perspective. The current development team at QAware GmbH consists of ten developers.

The system, at the center of the project, is being developed for a large German automotive company. It is mainly used by car dealerships and allows to research and finally order parts for a given vehicle, e.g. in the context of a repair. It was created in 2015 as a part of the renovation of one of its older predecessors. Until 2020, this system was operated and developed by another company. It was taken over by QAware GmbH as a part of a larger transition in the spring of 2020. It is to date estimated to have over 50.000 users worldwide.

The project has the following stakeholders:

- The business department that owns the system.
- The business departments that deliver the information that is used in the system, e.g. parts of vehicles.
- The business departments representing users, e.g. car dealerships in different countries.
- Applications and systems that use interfaces or information from the system.
- The development team.

4.2.2. Results from the List of Technical Debt Items

This section provides an overview of the results obtained from the list of TD items collected from the Candidate Project. The objective is to use these results to assess the proportions and impacts of different levels of TD in a project from practice.

The TD items in the list do not have a one-to-one mapping with TD types. This means that every TD item can belong to multiple categories of TD. This must be taken into account as a limitation when drawing conclusions. Figure 4.7 represents the proportions of different types of TD in the list.

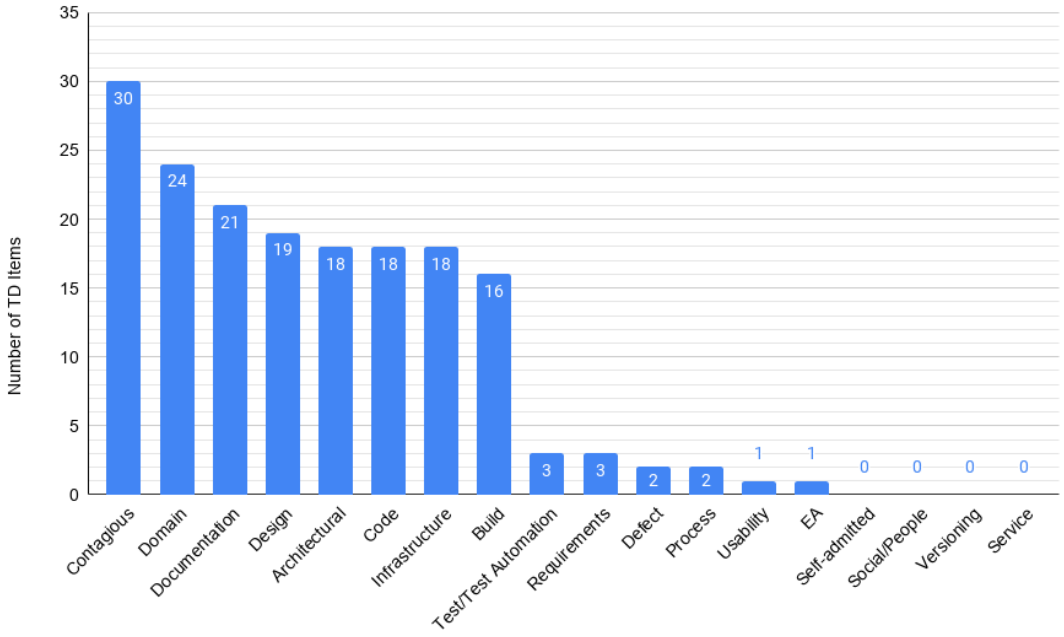

Figure 4.7.: The numbers of TD items that belong to certain types of TD. Each TD item in the list is linked to one or more types of TD.

TD Type/Criticality	Very High	High	Medium	Low	Very Low
Contagious	9	13	5	3	0
Domain	7	10.5	6.5	0	0
Documentation	1	13	7	0	0
Design	6	8	3	2	0
Architectural	3	10	4	1	0
Code	3	9	3	3	0
Infrastructure	9	4	3	2	0
Build	1	7	4	4	0
Test/Test Automation	2	0	0	1	0
Requirements	0	1	2	0	0
Defect	0	1	1	0	0
Process	0	1	1	0	0
Usability	1	0	0	0	0
EA	0	0	1	0	0

Figure 4.8.: A heat map representing the distribution of different types of TD on a scale of criticality. The cells represent the observed frequencies.

TD Type/Principal	Very High	High	Medium	Low	Very Low
Contagious	6	6	11	7	0
Domain	6	5	8	5	0
Documentation	4	4	7	5	1
Design	5	3	6	5	0
Architectural	2	7	5	4	0
Code	4	2	6	6	0
Infrastructure	2	5	5	6	0
Build	0	6	4	5	1
Test/Test Automation	1	1	1	0	0
Requirements	0	0	1	2	0
Defect	0	0	2	0	0
Process	0	0	1	1	0
Usability	0	0	0	1	0
EA	0	0	0	1	0

Figure 4.9.: A heat map representing the distribution of different types of TD on a scale of the cost of the principal. The cells represent the observed frequencies.

TD Type/Interest	Very High	High	Medium	Low	Very Low
Contagious	3	10	10	7	0
Domain	2	8	8	6	0
Documentation	0	5	11	4	0
Design	4	5	7	3	0
Architectural	1	5	7	5	0
Code	3	5	5	4	0
Infrastructure	2	6	5	5	0
Build	0	3	5	8	0
Test/Test Automation	0	3	0	0	0
Requirements	0	1	1	1	0
Defect	0	1	1	0	0
Process	0	0	0	2	0
Usability	0	1	0	0	0
EA	0	1	0	0	0

Figure 4.10.: A heat map representing the distribution of different types of TD on a scale of the cost of the interest. The cells represent the observed frequencies.

The cost of a TD item is expressed through its criticality, principal, and interest. Each of them is rated on a five-point scale in the list. Figure 4.8 expresses the relationship between TD types and their criticality, Figure 4.9 expresses the relationship between TD types and the cost of their principal, and Figure 4.10 expresses the relationship between TD types and the cost of their interest in the list. The stakeholders, who pay the interest on the TD item, are represented in Figure 4.11. Figure 4.12 provides an overview of the

effects that the different types of TD have on the project and its stakeholders.

The number and the respective intensity of the red color in the cells of these heat maps indicate the observed amount of TD items of a certain type. As can be seen from these maps, there is not enough data between the TD types "Build" and "EA" to draw clear conclusions. TD types that are not present in the list of TD items have been omitted from the heat maps.

TD Type/Payer	Product Owner	Users	Operations	Development
Contagious	1	15	15	12
Domain	11	12	8	5
Documentation	7	2	5	15
Design	1	9	10	6
Architectural	1	7	10	7
Code	2	7	7	9
Infrastructure	1	7	9	9
Build	1	2	4	10
Test/Test Automation	0	2	2	3
Requirements	1	2	0	0
Defect	0	1	1	0
Process	0	0	1	1
Usability	0	1	1	0
EA	0	1	0	0

Figure 4.11.: A heat map representing the distribution of different types of TD among the payers of the interest. The cells represent the observed frequencies.

TD Type/Effect	Productivity	Maintainability	Quality	Stability	Performance	Security	Usability	Diagnosability
Contagious	11	12	13	13	7	1	0	1
Domain	11	6	9	6	6	6	3	1
Documentation	18	15	14	2	0	2	1	0
Design	5	7	7	8	6	2	0	1
Architectural	5	9	7	8	2	1	0	2
Code	6	6	6	6	5	4	0	0
Infrastructure	11	5	5	9	2	4	0	2
Build	10	12	7	1	1	2	0	0
Test/Test Automation	2	2	3	2	0	0	0	0
Requirements	1	0	2	0	1	1	3	0
Defect	0	0	2	2	0	0	0	0
Process	2	1	1	1	0	0	0	0
Usability	0	0	1	1	1	0	0	0
EA	0	0	1	0	1	0	1	0

Figure 4.12.: A heat map representing the different types of TD and their effects on the project. The cells represent the observed frequencies.

A subset of the items in the list of TD were also tracked in issue tracking software Jira. This information about task sizes and mitigation efforts was used to supplement the

data in the list, in particular the principal of the items. Some of this data was also used to evaluate the cost of the instances of domain level TD in the previously presented case studies. The task sizes in Jira are rated based on T-shirt sizes, explained in Table 3.1. Data from Jira about task sizes of TD items, representing their principal, is visualized in Figure 4.13. The number of the TD item matches the order numbers of TD items in the sanitized list of TD items [54]. The numbers in the cells represent the amount of tasks with the corresponding task size that belong to a certain TD item. Some of the tasks can be matched to multiple TD items.

TD Item	XS	S	M	L	XL
1				1	
3				2	
15			1		4
17		1		3	2
25					1
26					1
30			1		4
39					2
52				1	
59		1		1	
64					2
67			1	1	
69				1	
76			1		
80	1			1	
87	1	1		1	2
4,55,68,70					1
1,8				2	7

Figure 4.13.: Data about task sizes of TD items collected from Jira. The task sizes represent the principal of the TD item. The numbers in the cells represent the amount of tasks with the corresponding task size that belong to a certain TD item.

5. Discussion of Results

This chapter analyzes and discusses the results of the studies and answers the Research Questions, listed in section 1.1. First, it discusses the characteristics and effects of domain level TD in practice; second, proposes a checklist for identifying domain level TD; and third, discusses the proportions and impacts of different levels of TD in the Candidate Project. Finally, limitations of the study are listed.

5.1. Characteristics and Effects of Domain Level Technical Debt in Practice (RQ1)

The empirical inquiries based on case studies conducted in this thesis make it possible to observe the characteristics and effects of domain level TD in practice and thus provide an answer to RQ1. This in turn provides the means to work on a clearer understanding of the phenomenon.

Table 5.1 summarizes the characteristics and effects of domain debt that were identified in the case studies and compares the different cases to each-other. The dimensions of characteristics and effects in the table were developed based on previous literature and information collected from the case studies.

5.1.1. Dimensions of Characteristics and Effects in the Case Studies

From [5, 10, 9, 19, 12] we have adopted the following dimensions:

- High interest - the interest rate of the debt, paid at every time point "x" during the existence of the debt, is considered "high" on a scale from "low", "medium" to "high".
- High principal - the principal of the debt, paid to fully eliminate the debt from the system, is considered "high" on a scale from "low", "medium" to "high".
- High criticality - the criticality of the debt was rated "high". It is based on the probability of the interest of the debt and further described in subsection 3.2.1.

From [11] we have adopted the following dimension:

- Contagious - the debt is considered to be contagious, i.e. it grows and spreads with time.

Characteristic or Effect/Case Study	CS1 - Multi-tenant	CS2 - Database Problem	CS3 - Interactive Filtering	CS4 - GDPR	CS5 - Ground Truth	CS6 - Crawler Attacks
High interest	x	x	x	x	x	x
High principal	x	x	x	x	x	
High criticality	x	x	x	x	x	x
Contagious	x	x	x	x	x	
Inadvertent	x	x	x	(x)	x	(x)
Deliberate				(x)	x	
Prudent	x	(x)	x			
Reckless		(x)		x	x	(x)
Domain knowledge required	x	x	x	x	x	(x)
Dependent on the domain	x	x	x	x	x	x
Independent of the technology	x	x	x	x	x	
Repayment requires more than redesign	x			x	x	
Developers not main payers		x	x	x	x	x
Not able to fulfill use cases	x	x	x	x	x	
Cause: lack of initial design and planning	x	x	x			x
Cause: lack of vision and ownership	x			x	x	
Cause: conflict with agile	x		x			
Technology misused	x	x	x			
Also social/people TD	x	x	x	x	x	x
Also design TD or architectural TD	x	x	x			x
Also versioning TD					x	
Indicator: low performance		x	x			x
Indicator: user complaints/feedback		x	x			x
Indicator: code/design						x

Table 5.1.: A comparison of the domain level TD case studies based on characteristics and effects identified from previous literature and from the case studies themselves. “x” means that the characteristic applies to the case study. “(x)” means that the characteristic may apply to the case study, but it is not entirely clear.

From [26] we have adopted the following four dimensions:

- Inadvertent
- Deliberate
- Prudent
- Reckless

From [2] we have adopted the following dimensions:

- Domain knowledge required - certain domain knowledge is required to be able to identify, understand, and/or manage the debt.
- Dependent on the domain - the debt only becomes visible when it is considered in combination with the domain of the system. This means that the system has to be active in its domain, i.e. actively used, for the debt to be able to take effect.
- Independent of the technology - the debt does not depend on the technology and cannot be detected directly from implementation level artifacts, such as the source code of the system. The technology on its own does not cause the debt.
- Repayment requires more than redesign - repaying the principal of the debt does not only require making changes to the system, but also requires more extensive changes, such as organizational or cultural changes.
- Developers not main payers - the main payers of the interest of the debt are the users of the system and/or the business/operations departments. The development team is affected secondarily.
- Not able to fulfill use cases - the debt makes it impossible for the system to properly fulfill its actual use cases and causes usability issues.
- Cause: lack of initial design and planning - an insufficient or non-existent initial planning and design phase is a central cause of the debt.
- Cause: lack of vision and ownership - missing shared far-sighted vision among stakeholders or the lack of willingness to take responsibility are a central cause of the debt.
- Cause: conflict with agile - the debt is partially caused by principles of agile methodologies.

The following dimensions were detected based on information gathered from the case studies:

- Technology misused - the system implements technologies that are unsuitable for the tasks assigned to it.
- Also social/people TD - the debt can also be considered as social/people TD [10, 28, 30, 12]. Some of the items of debt in the case studies have aspects that can be linked to inefficient social structures.

- Also design TD or architectural TD - the debt can also be considered to be a structural TD, such as design TD [2, 10, 28, 29, 12], or architectural TD [10, 28, 29, 12]. Some of the items of debt in the case studies were linked to system level design.
- Also versioning TD - the debt can also be considered as versioning TD [10, 29, 12]. One of the items of debt in the case studies can be linked to deficiencies in version control.
- Indicator: low performance - low system performance is an indicator of the debt.
- Indicator: user complaints/feedback - complaints and feedback from the users of the system are an indicator of the debt.
- Indicator: code/design - implementation level code and design issues are an indicator of the debt, e.g. long methods or missing security checks.

5.1.2. Analysis of Characteristics and Effects in the Case Studies

Almost all instances of domain debt in the case studies were relatively expensive and with rather extensive consequences. This is expressed by the relatively high interest rate and principal of the debts that in all of the cases led to a criticality rating of "high". The only case where the principal was not considered high was CS 6: Crawler Attacks, which dealt with web crawling attacks on a system.

In five cases the debt was contagious and had spread all over the system. In three of the cases, its elimination did not only require redesigning and refactoring the system, but also required organizational and cultural changes.

In most cases certain domain knowledge is required to understand the scope of the debt, i.e. its causes, its consequences, and how to eliminate it. Only in the case of CS 6: Crawler Attacks can one argue whether a significant amount of domain knowledge is needed to understand the relevance of crawling attacks. Domain debt seems to be relatively independent from the technology and, unlike other types of TD, sometimes only becomes visible in the context of the domain, especially when it is actively used and executed. The implementation of the system itself is not necessarily flawed or incorrect, but it is simply used for purposes for which the underlying technologies are not really intended for. In three cases it can be said that the technology was being misused. Only the debt in CS 6: Crawler Attacks can be detected directly by taking a look at the implementation of the system, given that protection against crawling attacks is somewhat general knowledge. Thus, it is possible to see a connection between domain debt and its independence from the technology and source code.

Domain debt is often due to a lack of design and poor planning in the early stages of a project, as can be seen in four cases out of six. As a result, it often lies in the foundation of the system and this may partly explain why the debt is often so expensive.

Short initial planning phases are in vogue today and popular agile methodologies, such as Scrum and Kanban, support the principle of continuous learning as you go. There seems to be a slight conflict between being fully agile and avoiding domain debt and in two of the six cases the cause of the debt can be linked to agile principles. Some fundamental design decisions cannot be redone so easily in later phases and should be done correctly from the beginning. To start full-scale development, there should first be a certain level of domain knowledge in the team. Another potential cause for domain debt is a lack of vision and ownership. This is the case in three case studies, where team members lack a shared far-sighted vision and avoid taking responsibility for certain problems. Thus, domain debt can also arise from organizational deficiencies. This might partially explain why the repayment of domain debt often also requires cultural and organizational changes. The repayment of the principal of these three cases, potentially caused by a lack of ownership and vision, requires more than just redesigning the system, i.e. also organizational changes.

Domain debt is to some extent inadvertent in all of the observed cases. Only in two cases could one also take a deliberate dimension into consideration. Namely in CS 4: GDPR, dealing with GDPR compliance, where a possible cause was that the correct domain design was discounted in order to achieve other conflicting objectives, and in CS 5: Ground Truth, where the debt was inadvertent at first, but was later deliberate due to a lack of initiative to solve the problem. It is possible to see a slight connection between the debt being deliberate and its cause being a lack of ownership and vision. The observed instances of domain debt are fairly evenly distributed between the reckless and prudent dimensions of the TD quadrant.

Instances of domain debt can sometimes be seen as combinations of different types of TD and at first sight, it might not be clear which of them is most suitable for categorizing the issue. Domain debt is often associated with the system level design and thus, it has features of structural debt, such as architectural TD or design TD. Architectural TD and design TD were considered in four of the cases. In addition to that, a recurring cause of the debt is that there is a lack of knowledge, expertise, and experience in the development team. This can be linked to social/people TD, which includes bad knowledge distribution or lack of team members with necessary expertise. In all of the six cases the debt was associated with social/people TD. Versioning TD was considered in one case, but this occurrence seems to be a coincidence. It might be useful to use all of these classifications in combination, as it helps describe different aspects of the debt.

In five out of six cases, the main interest payers were the users of the affected system, followed by other stakeholders, such as the operations, sales, and marketing departments. This indicates that those instances of domain debt resulted in usability issues that were often invisible to the developers. This is partially due to the fact that in five out of six cases the debt made it impossible for the system to fulfill its intended use cases, i.e. the users could not use it the way they were expecting to.

Smells that indicate the existence of a debt in the case studies were the system's

low performance in three cases, user complaints and feedback in three cases, and code/design in one case. Code and design served as an indicator in CS 6: Crawler Attacks. The other two types of smells were always present in combination with each-other.

In conclusion, the answer to RQ1 is that, based on the case studies, instances of domain level TD in practice are often expensive, highly critical, inadvertent, and dependent on the domain. Systems that contain domain debt are sometimes unable to fulfill certain use cases and therefore particularly affect the users and business departments. Domain knowledge tends to play an important role in the life cycle of domain debt. Domain debt sometimes has close ties to architectural, design, and social/people TD. As soon as the debt starts to become more dependent on the technology and when it can be directly detected from implementation level artifacts without a need of domain understanding, it tends to move away from being domain level TD. But it must be noted that such a small amount of empirical evidence is insufficient to make any tenable generalizations with regard to domain debt.

5.2. Identifying Domain Level Technical Debt (RQ2)

The goal of RQ2 is to understand if and how domain level TD could be identified in a system. The previously discussed characteristics, indicators, and causes of domain debt form a basis for being able to put together a checklist that can be used to determine whether an item of TD belongs to the category of domain level TD. Ideally, an analyst could use this checklist or questionnaire as a basis for a procedure for identifying and quantifying the amount of domain debt in any system, for instance during an analysis phase of a transition. This procedure would be a first step towards being able to automatically measure domain debt.

Due to the abstract nature of domain debt and due to the relevance of domain knowledge, automated detection and measurement are most likely impossible today. To be able to do this, the automated analysis tool should be able to model and analyze the domain of the system. This means that the tool would have to have a mind of its own that is able to somehow model the current domain and state of the system and compare this information with the target domain and state of the system. Technologies that could possibly form a pathway to being able to do this are Artificial Intelligence and Big Data Analytics.

The following is a checklist designed to simplify the manual identification and measurement of domain level TD in practice. It has been compiled on the basis of information gathered from previous literature and the previously presented case studies. The more questions that can be answered with “yes”, the greater the probability that the item of TD in question belongs to the category of domain level TD. We propose four categories of weights for the questions, which were determined on the basis of the case

studies. It is important to keep in mind that the following questions, categories, and weightings need further confirmation in the future by conducting a more comprehensive study.

The weights for the questions were selected by experimenting with samples of debt from the case studies and the list of TD items, which is presented in subsection 4.2.2. Both examples related to domain level TD and examples not related to domain level TD were used to determine the weights and the threshold.

5.2.1. Domain Debt Identification Checklist

The purpose of this checklist is to identify items of TD that belong to the category of domain level TD. It can be used for debt analysis and classification and allows to look at TD from a domain perspective.

The questions have been divided into four categories with different weights, depending on how closely they are linked to domain level TD. Each question that can be answered with "yes" gives a certain weight. If the total sum of weights is greater than or equal to the threshold below, there is a high probability that the TD item in question belongs to domain level TD.

Explanation	Weight
Domain level TD threshold.	50
The maximum possible amount of weights.	153
The TD item in question probably does **not** belong to domain level TD.	0 - 49
The TD item in question probably belongs to domain level TD.	50 - 153

General Questions

The purpose of the general questions is to first ask trivial questions in order to understand whether the problem in question is a TD at all and what its estimated cost is. The cost estimates are used in the following categories of questions.

- Is the problem in question an item of technical debt (TD)?
  (If "yes", proceed. The checklist is meant for assessing items of TD.)
- How high is the interest of the debt?
  (The interest rate of the debt, paid at every time point "x" during the existence of the debt.)
  - "very low"/"low"/"medium"/"high"/"very high"
- How high is the principal of the debt?
  (The principal of the debt, paid to fully eliminate the debt from the system.)
  - "very low"/"low"/"medium"/"high"/"very high"

- How high is the criticality of the debt?
  (The criticality of the debt represents the probability of the interest having to be paid and consequently how soon the debt should be removed from the system.)
  - "very low"/"low"/"medium"/"high"/"very high"

First Order Questions

The following questions have a very strong correlation with domain level TD, as they are part of its definition. Each question that can be answered with "yes" adds a weight of **40**.

Nr.	Question	Weight
1	Is domain knowledge needed for being able to identify, understand, and manage the debt? *(E.g. repayment of the debt's principal requires understanding certain specifics of the discipline or field from the domain.)*	40
2	Is the debt dependent on the domain of the system? *(I.e. the debt only becomes visible, when the system is considered in the context of its domain and the debt can be represented as a quality gap between the system and its domain. The system is functional, i.e. it can "do something", but does not meet the requirements from its domain.)*	40

Second Order Questions

The following questions have a strong correlation with domain level TD. They describe the core nature of domain level TD that distinguishes it to a great extent from other types of TD. Each question that can be answered with "yes" adds a weight of **10**.

Nr.	Question	Weight
3	Is the debt independent from the technology? *(I.e. it is impossible to detect the debt directly from implementation level artifacts, e.g. the source code of the system.)*	10
4	Are the main interest payers of the debt the users of the system, the product owners, and/or the sales/marketing/operations departments of the organization? *(I.e. the debt does not necessarily affect the developers as would other types of TD. E.g. the debt causes usability problems that mainly affect users.)*	10
5	Does the debt cause the system to fail to fulfill its actual intended use cases? *(I.e. the system or the part of the system affected by the debt cannot be used as expected.)*	10

6	Does the system have to be actively used and executed in order for the consequences of the debt to materialize? *(I.e. so that the interest will have to be paid.)*	10
7	Does the repayment of the debt's principal require more than making changes to the system itself? *(I.e. additional organizational/cultural changes or changing neighboring systems.)*	10

Third Order Questions

The following questions have a medium correlation with domain level TD. They describe aspects that often accompany domain level TD and are not that common for other types of TD. Each question that can be answered with "yes" adds a weight of **3**.

Nr.	Question	Weight
8	Are the systems technologies being misused? *(I.e. the implementation and design of the system alone are correct, but are being used for inappropriate purposes.)*	3
9	Does the debt have links to social/people TD? *(I.e. is it resulting from bad knowledge distribution or lack of team members with necessary expertise.)*	3
10	Does the debt affect fundamental design/architecture layers of the system? *(I.e. whether the system's design/architecture needs to be changed to repay the debt's principal.)*	3
11	Is low system performance a common indicator of the debt?	3
12	Are user complaints and feedback common indicators of the debt?	3

Fourth Order Questions

The following questions have a lower correlation with domain level TD. They describe aspects that often accompany domain level TD, but frequently also occur with other types of TD. Each question that can be answered with "yes" adds a weight of **1**.

Nr.	Question	Weight
13	Is the debt inadvertent? *(I.e. it has not been achieved through deliberate planning, but was rather unintentional.)*	1
14	Is a lack of initial design and planning or requirements analysis a potential cause of the debt?	1
15	Is a lack of vision and ownership among the stakeholders of the system a potential cause of the debt?	1

16	Are principles of agile methodologies a potential cause of the debt? *(I.e. the principle of learning as you go has led to the debt. The issue is too large and fundamental to be easily corrected in later iterations.)*	1
17	Is the interest of the debt rated "high" or "very high"?	1
18	Is the principal of the debt rated "high" or "very high"?	1
19	Is the criticality of the debt rated "high" or "very high"?	1
20	Is the debt contagious by nature? *(I.e. does it spread with time.)*	1

5.2.2. Validation of the Checklist

The checklist and specifically its questions and wording were validated by eight senior software engineers at QAware GmbH. The structure of the validation procedure is described in subsection 3.2.2.

One of the goals of the validation procedure was to determine, if the questions are understandable. Most of the questions were considered understandable. Additional clarifications and, if necessary, examples were added to the questions that were not clear enough. In particular, it was considered that the questions in the first and second order should be clarified.

The evaluators found that the questions are purposeful and abstract enough to be able to apply them in a generalized way to different situations. It was mentioned that the checklist should remain compact and should not become too long and complicated. In general, the checklist was considered helpful when categorizing TD.

It was considered useful for being able to find out the source of the problem and what to do to get rid of the debt. Also, it was considered useful for being able to generally recognize that a problem is not always just of a technical nature. It was mentioned that the checklist can also be used in a retrospective analysis, to understand how the problem occurred and how to avoid it in the future.

One evaluator found that the fourth order questions could be removed from the list, because they are not closely linked to domain debt. Besides, one evaluator suggested that the checklist could also include negative-weighted questions, i.e. questions that contradict to domain level TD. These two suggestions were not implemented. First, because the fourth order questions are designed to guide the user of the checklist to evaluate different aspects of the problem. Even if they are not very significant, we believe they can add value. Second, adding questions with negative weights would require more research examining the characteristics of other types of TD. This remains something for the future.

The following are some additional questions that were suggested by the evaluators, but were not added to the checklist, because they are not backed up by the results from

the case studies:

- Does the debt make onboarding of new team members difficult and lengthy?
- Does the debt lead to strange terminology or even different terminology between users and developers?
- Does the debt hinder using the system in a slightly different context?

5.3. Proportions and Impacts of Different Levels of Technical Debt (RQ3)

The results, presented in section 4.2, make it possible to discuss the proportions and impacts of different levels of TD in a software project from practice and through it to provide an answer to RQ3. Due to the size of the list of TD items and the large differences in the sample sizes of different types of TD, we decided to group the debts for analysis.

Types of TD that were not present in the list of TD are omitted from the analysis. In addition, also the types usability TD, defect TD, and EA TD are omitted, because they do not fit into groups with other types and were marginally represented in the list. The following more abstract levels of TD were compared to each-other:

- Structural TD - includes architectural TD and design TD. Both address the system level design and structure of the system.
- Documentation Level TD - includes documentation TD and requirements TD. Both address non-code-related artifacts in a software project, such as requirements and code documentation. This level of software artifacts has also been mentioned in [55].
- Contagious TD - includes contagious TD separately. It is a more abstract characteristic-based type of TD that by definition includes several other types of TD.
- Domain Level TD - includes domain level TD separately. Since we pay special attention to domain debt in this thesis, we observe it separately.
- Environmental TD - includes infrastructure TD, build TD, and process TD. Environmental TD is by definition a supertype of those subtypes.
- Code Level TD - includes code TD and test/test automation TD. Both are closely related to code-related artifacts. This level of software artifacts has also been mentioned in [55].

The level contagious TD will not be taken into account in the following Chi-Square tests, because, unlike other levels of TD, it represents a characteristic-based perspective and is not well-comparable with others.

Proportions of Technical Debt

After applying the grouping to the data, we achieve relatively comparable data sets (see Figure 5.1). The most significant levels of debt are structural TD and environmental TD. Code level TD is the smallest level in the list. However, it is worth bearing in mind that since the list of TD was mainly compiled during initial analysis phases of a transition, more important issues were probably added to it. This means that the list is not complete and smaller code-related issues might be missing.

Around 34% of the 87 items of TD are considered to be contagious. Over a quarter of the items on the list have a correlation with domain level TD.

When taking a look at the more granular distribution, presented in Figure 4.7, we can see that the four most common types of TD are contagious TD, domain level TD, documentation TD, and design TD.

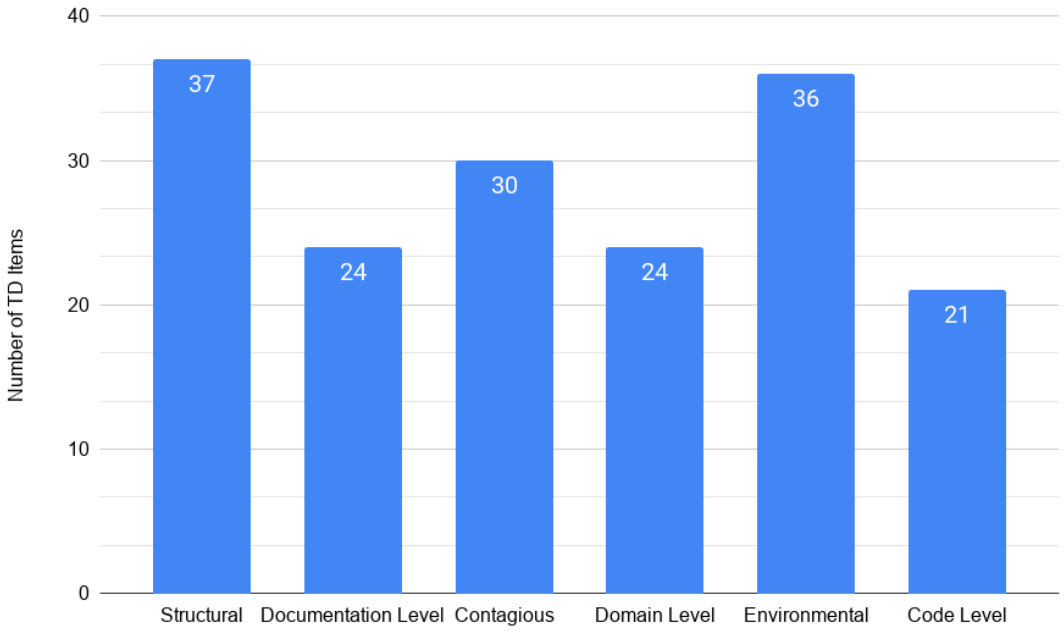

Figure 5.1.: Proportions of different levels of TD in the Candidate Project.

Criticality of Technical Debt

When applying the Chi-Square test for independence on the contingency table of the criticality, represented in Figure 5.3, it first has to be abstracted to fulfill the preconditions for the test, i.e. no more than 20% of the expected frequencies are less than 5 and none are less than 1 [41]. The column “Low” is added to “Medium” and the row “Contagious”

and the empty column "Very Low" are disregarded. Thus, the Chi-Square test is calculated for "Very High" to "Medium". After applying the test we achieve the following results:

$$\chi^2_{0,05}(8) = 15,507$$

$$\chi^2(8) = 7,94315 < 15,507$$

Since χ^2 is smaller than the critical value $\chi^2_{0,05}$, the evidence is apparently not strong enough to suggest an effect exists in the population.

The criticality ratings of different levels of TD follow a very similar pattern and this is clearly visible in Figure 5.2. All levels have a peak at "High", but some have a flatter distribution than others. Structural TD, documentation level TD, contagious TD, and domain level TD all have a relatively sharp peak at "High", whereas environmental TD and code level TD tend to have a more even distribution. Code level TD has two peaks, one at "High" and a smaller one at "Low". Documentation level TD tends to cause the least items of TD with a criticality of "Very High", compared to the other levels.

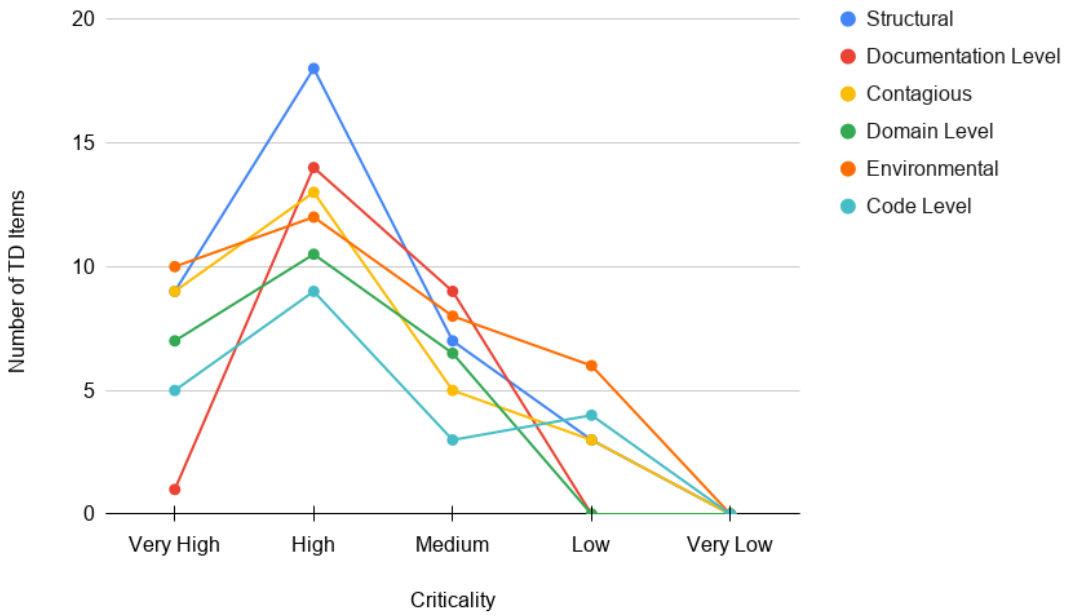

Figure 5.2.: A chart representing the criticality of different levels of TD in the Candidate Project.

TD Level/Criticality	Very High	High	Medium	Low	Very Low	SUM
Structural	9	18	7	3	0	**37**
Documentation Level	1	14	9	0	0	**24**
Contagious	9	13	5	3	0	**30**
Domain Level	7	10.5	6.5	0	0	**24**
Environmental	10	12	8	6	0	**36**
Code Level	5	9	3	4	0	**21**
SUM	**41**	**76.5**	**38.5**	**16**	**0**	**344**

Figure 5.3.: A contingency table representing the observed absolute frequencies of the criticality of different levels of TD in the Candidate Project.

Principal of Technical Debt

When applying the Chi-Square test for independence on the contingency table of the cost of the principal, represented in Figure 5.5, it first has to be abstracted to fulfill the preconditions for the test. The column "Very Low" is added to "Low" and the row "Contagious" is disregarded. Thus, the Chi-Square is calculated for "Very High" to "Low". After applying the test we achieve the following results:

$$\chi^2_{0,05}(12) = 21,026$$

$$\chi^2(12) = 8,52681 < 21,026$$

Since χ^2 is smaller than the critical value $\chi^2_{0,05}$, the evidence is apparently not strong enough to suggest an effect exists in the population.

The principal of different levels of TD has a more interesting distribution. As we can see in Figure 5.4 domain level TD, documentation level TD, code level TD, and contagious TD tend to follow a very similar pattern. They have a clear peak at "Medium" and tend to flatten between "High" and "Very High". It seems that domain level TD and code level TD even have a small increase in the number of TD items, when getting closer to a "Very High" principal. This trend is partly consistent with the previously made observations with regard to domain level TD, namely that it has a relatively high principal. Structural TD also has a peak at "Medium", but compared to the previous levels of TD, it has a very uniform distribution with a large arc. Environmental TD has an M-shaped distribution, with one peak at "High" and the other one at "Low". Compared to the others, it has significantly less items of TD with a "Very High" principal.

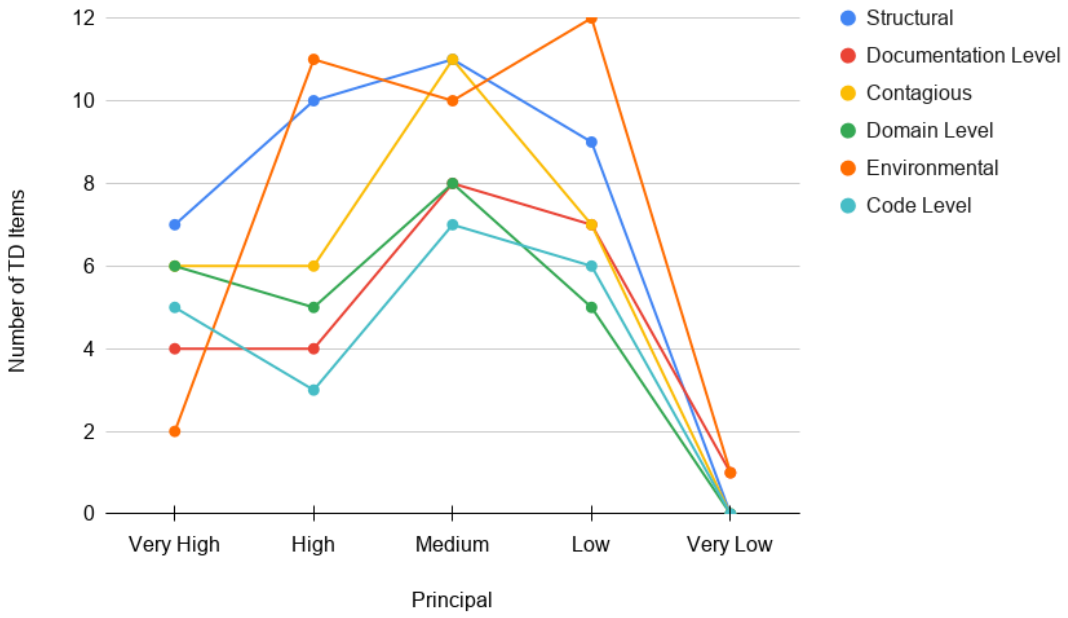

Figure 5.4.: A chart representing the size of the principal of different levels of TD in the Candidate Project.

TD Level/Principal	Very High	High	Medium	Low	Very Low	**SUM**
Structural	7	10	11	9	0	**37**
Documentation Level	4	4	8	7	1	**24**
Contagious	6	6	11	7	0	**30**
Domain Level	6	5	8	5	0	**24**
Environmental	2	11	10	12	1	**36**
Code Level	5	3	7	6	0	**21**
SUM	**30**	**39**	**55**	**46**	**2**	**344**

Figure 5.5.: A contingency table representing the observed absolute frequencies of the size of the principal of different levels of TD in the Candidate Project.

Interest of Technical Debt

When applying the Chi-Square test for independence on the contingency table of the cost of the interest, represented in Figure 5.7, it first has to be abstracted to fulfill the preconditions for the test. The column "Very High" is added to "High" and the row "Contagious" and the empty column "Very Low" are disregarded. Thus, the Chi-Square is calculated for "High" to "Low". After applying the test we achieve the following results:

$$\chi^2_{0,05}(8) = 15,507$$

$$\chi^2(8) = 10,0912 < 15,507$$

Since χ^2 is smaller than the critical value $\chi^2_{0,05}$, the evidence is apparently not strong enough to suggest an effect exists in the population.

The interest of the different levels of TD, visualized in Figure 5.6, tends to follow a similar pattern in most cases. Structural TD and documentation level TD both have a pyramid-shaped distribution with a very clear peak at "Medium". Compared to structural TD, documentation level TD rarely has items of TD with a "Very High" interest. Domain level TD and contagious TD have a much flatter distribution, with a peak that stretches through "High" and "Medium". Code level TD and environmental TD do not follow the pattern of the previously mentioned levels of TD. Code level TD has a peak at "High", followed by a flat decline towards lower interest rates. Environmental TD has a very clear peak at "Low", but often also has items of TD with a "Medium" and "High" interest.

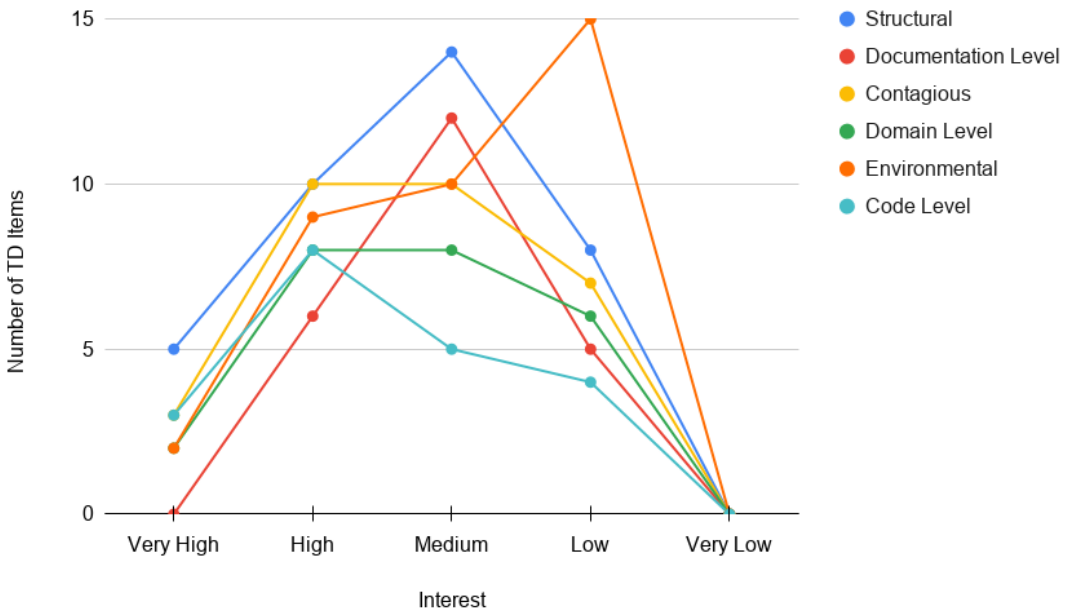

Figure 5.6.: A chart representing the size of the interest of different levels of TD in the Candidate Project.

TD Level/Interest	Very High	High	Medium	Low	Very Low	SUM
Structural	5	10	14	8	0	**37**
Documentation Level	0	6	12	5	0	**23**
Contagious	3	10	10	7	0	**30**
Domain Level	2	8	8	6	0	**24**
Environmental	2	9	10	15	0	**36**
Code Level	3	8	5	4	0	**20**
SUM	**15**	**51**	**59**	**45**	**0**	**340**

Figure 5.7.: A contingency table representing the observed absolute frequencies of the size of the interest of different levels of TD in the Candidate Project.

Interest Payers

When applying the Chi-Square test for independence on the contingency table of the interest payers, represented in Figure 5.9, it first has to be abstracted to fulfill the preconditions for the test. The row "Contagious" is disregarded from the test. After applying the test we achieve the following results:

$$\chi^2_{0,05}(12) = 21,026$$

$$\chi^2(12) = 36,7904 > 21,026$$

Since χ^2 is larger than the critical value $\chi^2_{0,05}$, an association between TD level and interest payers was observed. Figure 5.10 compares the contributions to the Chi-Square statistic to see which variables have the largest Chi-Square values that may indicate dependence.

Figure 5.8 visualizes the interest payers of different levels of TD. The distributions are noticeably different in all cases.

Structural TD tends to frequently cause consequences for the operations department, but also the users and developers often pay the interest. Structural TD includes architectural TD and design TD. Figure 4.11 shows that they are both relatively similar, when considering interest payers.

Documentation level TD mainly affects the developers, but often also the product owners. This is understandable, because documentation level artifacts are mainly used by internal stakeholders, specifically by the developers. Documentation level TD includes requirements TD and documentation TD. When taking a look at Figure 4.11, it can be seen that instances of requirements TD affect the users and the product owner, whereas documentation TD is mainly influencing development processes. Figure 5.10 indicates that the rather large influence on product owners has a slight contribution to the Chi-Square statistic.

Contagious TD primarily affects the users, the operations department, and the development team, but rarely the product owners.

Domain level TD tends to mainly have an effect on the users and the product owners and often also on the operations department. Developers are less frequently affected by domain level issues and this partly also confirms the observations made earlier in this thesis. Figure 5.10 indicates that the large influence on product owners and the small influence on developers has a significant contribution to the Chi-Square statistic. Domain level TD tends to have a noticeably different impact on project stakeholders compared to other levels of TD.

The interest of environmental TD is mainly paid by the developers, but often also by the operations department. Users are less likely to have to pay the interest on this debt and product owners are influenced in rare cases.

As expected, the interest of code level TD is predominantly paid by the development team, but often also by the operations department and the users. It rarely has an effect on the product owners.

When considering the aggregated results from Figure 4.11, developers were interest payers for 89% of the total items, the operations department was the interest payer for 84% of the total items, users were the interest payers for 78% of the total items, and product owners were interest payers for 30% of the total items.

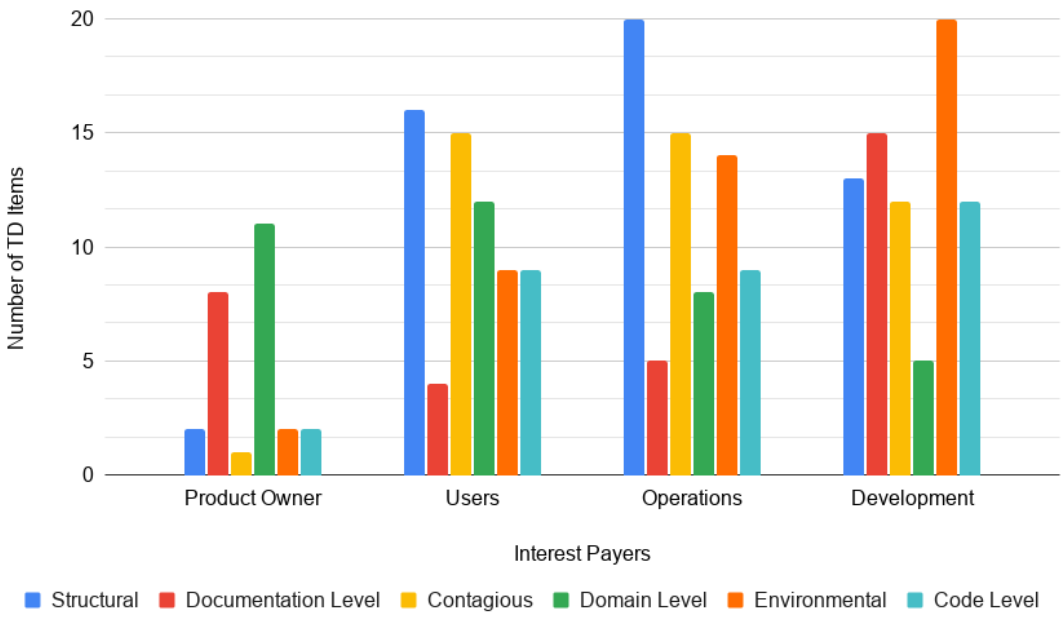

Figure 5.8.: A chart representing the interest payers of different levels of TD in the Candidate Project.

TD Level/Payer	Product Owner	Users	Operations	Development	SUM
Structural	2	16	20	13	**51**
Documentation Level	8	4	5	15	**32**
Contagious	1	15	15	12	**43**
Domain Level	11	12	8	5	**36**
Environmental	2	9	14	20	**45**
Code Level	2	9	9	12	**32**
SUM	**26**	**65**	**71**	**77**	**478**

Figure 5.9.: A contingency table representing the observed absolute frequencies of the interest payers of different levels of TD in the Candidate Project.

TD Level/Payer	Product Owner	Users	Operations	Development
Structural	3.12	0.687067	2.02241	0.905422
Documentation Level	3.76163	2.12326	1.87723	1.81417
Domain Level	8.94295	0.863673	0.507937	4.03279
Environmental	2.43669	0.535592	0.101587	1.72689
Code Level	1.06163	0.0857653	0.00223214	0.181476

Figure 5.10.: A heat map representing the Chi-Squared values for the interest payers of different levels of TD in the Candidate Project.

Interest Effects

When applying the Chi-Square test for independence on the contingency table of the interest effects, represented in Figure 5.12, it first has to be abstracted to fulfill the preconditions for the test. The row "Contagious" and the columns "Security", "Usability", and "Diagnosability" with small sample sizes are disregarded from the test. Thus, the Chi-Square is calculated for "Productivity" to "Performance". After applying the test we achieve the following results:

$$\chi^2_{0,05}(16) = 26,296$$

$$\chi^2(16) = 26,5134 > 26,296$$

Since χ^2 is slightly larger than the critical value $\chi^2_{0,05}$, an association between TD level and interest effects was observed. Figure 5.13 compares the contributions to the Chi-Square statistic to see which variables have the largest Chi-Square values that may indicate dependence.

Figure 5.11 provides an overview of the effects of different levels of TD in the Candidate Project. The four most common aggregated effects are related to productivity, maintainability, quality, and stability.

Structural TD tends to mainly affect the maintainability and stability of the system. This explains why the operations department is often the interest payer of structural TD, as they must ensure the stability of the system and, for example, deal with system outages. Figure 5.13 indicates that the influence on productivity has a slight contribution to the Chi-Square statistic.

Documentation level TD often affects productivity and quality. For instance, the lack of proper documentation reduces system understanding and this makes it more difficult to make changes to the system and thereby reduces productivity. In addition, it is more difficult to detect issues without documentation and this affects the quality of the system. Figure 5.13 indicates that the small influence on stability has a significant contribution to the Chi-Square statistic.

The effects of contagious TD involve stability, quality, maintainability, and productivity. As mentioned earlier, contagious TD is a characteristic-based aggregate of multiple other types of TD and this explains why it follows an aggregated pattern in the graph.

Domain level TD tends to mainly affect productivity and quality. This is to some extent caused by the fact that the TD items linked to domain debt are often also linked to documentation level TD and contagious TD in the list. Earlier we mentioned that domain debt often causes usability issues. If we take a look at "usability" in Figure 5.11, we can see that documentation level TD and domain level TD are the two levels of TD that most often cause usability issues.

Environmental TD often affects productivity and maintainability. When taking a look at the TD types in Figure 4.12, it can be seen that this is mainly caused by infrastructure TD and build TD. Together they affect the productivity in the project. This is understandable because, for example, slow infrastructure and build pipelines slow down progress. At the same time, maintainability is mainly affected by build TD. An example of this is the absence of integrated quality analysis tools in a build pipeline, which limits maintainability.

Finally, code level TD tends to have a wide-ranging impact and often affects quality, productivity, maintainability, and stability.

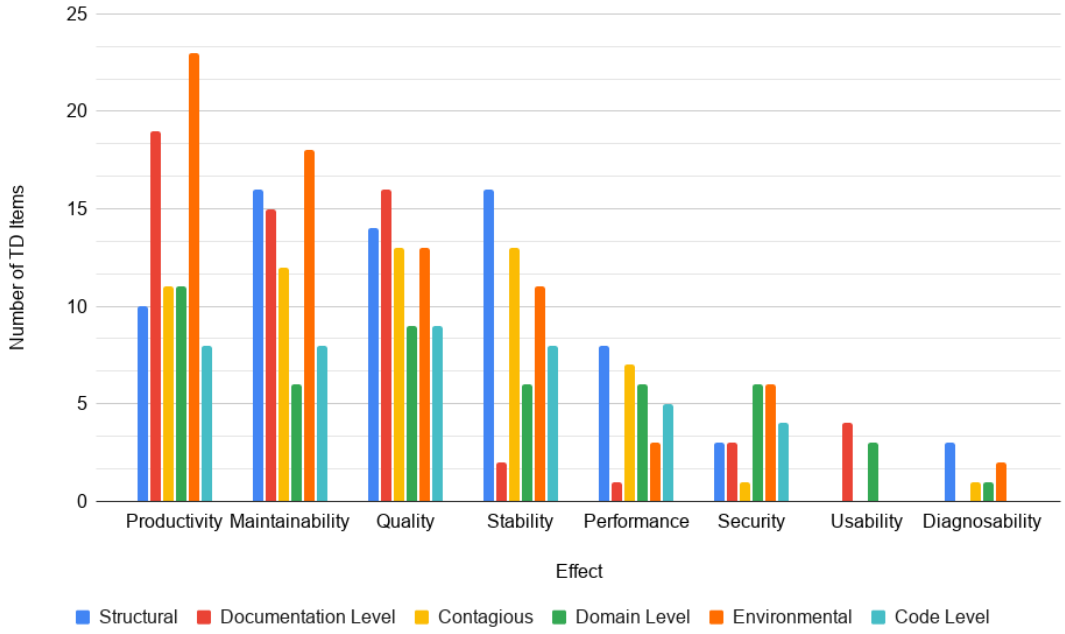

Figure 5.11.: A chart representing the interest effects of different levels of TD in the Candidate Project.

TD Level/Effect	Productivity	Maintainability	Quality	Stability	Performance	Security	Usability	Diagnosability	SUM
Structural	10	16	14	16	8	3	0	3	**70**
Documentation Level	19	15	16	2	1	3	4	0	**60**
Contagious	11	12	13	13	7	1	0	1	**58**
Domain Level	11	6	9	6	6	6	3	1	**48**
Environmental	23	18	13	11	3	6	0	2	**76**
Code Level	8	8	9	8	5	4	0	0	**42**
SUM	**82**	**75**	**74**	**56**	**30**	**23**	**7**	**7**	**477**

Figure 5.12.: A contingency table representing the observed absolute frequencies of the interest effects of different levels of TD in the Candidate Project.

TD Level/Effect	Productivity	Maintainability	Quality	Stability	Performance
Structural	3.1538	0.0197044	0.061338	2.82313	0.987673
Documentation Level	1.45642	0.380705	1.05385	5.1899	2.88461
Domain Level	0.042502	1.09723	0.00158844	0.0108424	2.09923
Environmental	1.09564	0.153289	0.52652	0.00368073	1.49426
Code Level	0.528417	0.149857	0.00158844	0.483303	0.814334

Figure 5.13.: A heat map representing the Chi-Squared values for the interest effects of different levels of TD in the Candidate Project.

TD Level	Criticality	Principal	Interest	Interest Payers	Interest Effects
Structural	Clear peak at “High”	Peak at “Medium”, uniformly distributed	Clear peak at “Medium”	Operations, development, users	Maintainability, stability
Documentation Level	Clear peak at “High”, rarely “Very High”	Peak at “Medium”	Clear peak at “Medium”, rarely “Very High”	Development, product owners	Productivity, quality (peculiarly usability)
Contagious	Clear peak at “High”	Peak at “Medium”	Peak at “High” and “Medium”	Users, operations, development	Stability, quality, maintainability, productivity
Domain Level	Clear peak at “High”	Peak at “Medium”, increase at “Very High”	Peak at “High” and “Medium”	Users, product owners	Productivity, quality (peculiarly usability)
Environmental	Peak at “High”, evenly distributed	Peaks at “High” and “Low”, rarely “Very High”	Peak at “Low”	Development, operations	Productivity, maintainability
Code Level	Peaks at “High” and “Low”	Peak at “Medium”, increase at “Very High”	Peak at “High”	Development, operations, users	Quality, productivity, maintainability, stability
Statistically significant?	No	No	No	Yes	Yes

Table 5.7.: A summary of the impacts of different levels of TD on the Candidate Project.

It was not possible to prove that the results from the list of TD items concerning the criticality, the cost of the principal, and the cost of the interest have any statistical significance. This is possibly caused by the fact that they were vaguely estimated on scales from "Very High" to "Very Low" and were not backed up enough by actual data on costs and sizes of issues. However, it was possible to prove statistical significance for the results concerning the interest payers and interest effects and therefore, they seem to depend on levels of TD.

A summary of the economic and technological impacts of different levels of TD in the Candidate Project is visualized in Table 5.7.

Around 43% of the issues in the list of TD items are linked to structural TD. Structural TD tends to have a "High" criticality, a "Medium" principal, and a "Medium" interest rate. Its interest is most often paid by the operations department, developers, and users. It typically affects maintainability and stability.

Around 28% of the issues in the list of TD items are linked to documentation level TD. Documentation level TD tends to have a "High" criticality, "Medium" principal, and "Medium" interest rate. It rarely has a "Very High" criticality or interest rate. Its interest is mainly paid by the developers and product owners. It usually affects productivity and quality. Unlike others, it often also affects usability.

Around 34% of the issues in the list of TD items are linked to contagious TD. Contagious TD tends to have a "High" criticality, "Medium" principal, and "High/Medium" interest rate. Its interest is often paid by the users, operations department, and developers. It typically affects stability, quality, maintainability, and productivity.

Around 28% of the issues in the list of TD items are linked to domain level TD. Domain level TD tends to have a "High" criticality, "Medium" and increasingly also "Very High" principal, and "High/Medium" interest rate. Its interest is often paid by the users and product owners. It usually affects productivity and quality. Unlike others, it often also affects usability.

Around 41% of the issues in the list of TD items are linked to environmental TD. Environmental TD tends to have a "High" criticality, "High" or "Low" principal, and "Low" interest rate. Its principal is rarely "Very High". Its interest is often paid by the developers and operations department. It mainly affects productivity and maintainability.

Around 24% of the issues in the list of TD items are linked to code level TD. Code level TD tends to have a "High" or "Low" criticality, "Medium" and increasingly also "Very High" principal, and "High" interest rate. Its interest is typically paid by the developers, operations department, and users. It often affects quality, productivity, maintainability, and stability.

5.4. Limitations of the Study

This section discusses the limitations and drawbacks of the studies conducted in this thesis from the perspectives of reliability and generalizability. The limitations require that the results of this thesis should be interpreted with caution. As is common for empirical and qualitative research, this study is affected by various constraints.

Reliability of the results:

- First, the characteristics of domain level TD in practice were analyzed based on six case studies. The case studies were based on data collected from questionnaires and interviews conducted with two senior software consultants. Therefore, the case studies rely on subjective opinions. However, both software consultants are well acquainted with the subject of this thesis and have previously studied the concept of domain level TD.
- Second, the list of TD items and specifically the categorization of those items relies on subjective opinions and estimates. However, it was compiled by four senior software architects and was later validated and complemented with data from issue tracking software. Also, a significant amount of effort was invested into compiling the list, as it formed the basis of taking over responsibility for the system during the transition.
- Third, the checklist for eliciting domain level TD is affected by the limitations of the case studies and needs wider validation in practice. The categorization and weigthing of the questions in the checklist is based on small-scale experimentation with items of TD from the case studies and the list of TD items. The checklist was empirically validated by eight senior software engineers from QAware GmbH, familiar with the concept of TD.

Generalizability of the results:

- First, the list of TD items and the case studies were collected and conducted in a single company, based in Germany. This means that the research might have cultural and organizational biases and does not take into account international specificities and the results may not be applicable to other organizations and countries.
- Second, the list of TD items was collected from only one project. Therefore, it is possible that due to the specifics of the project, the results of the study cannot be generalized to other projects with different contexts, clients, and structures.

6. Conclusion

This thesis maps and characterizes a new more higher level type of TD, called "Domain Level TD", which was first mentioned in 2019 [2]. Compared to higher level TD, it is often easier to analyze and measure lower implementation level TD, since it can mostly be detected automatically from code-related software artifacts. Yet the effects of higher level TD seem to be significantly more costly and far-reaching. An example of this is domain level TD that in some cases does not only affect the system, but the whole organization and can render the entire system useless, although its code may be of high quality.

This thesis summarizes a set of characteristics and effects of domain level TD in practice based on six case studies. These case studies serve as examples and describe instances of TD from practice that can be linked to domain level TD. The identified characteristics and effects allow to distinguish domain debt from other types of TD and to clarify its definition. The results indicate that domain debt tends to be relatively expensive and contagious. It is often dependent on the domain of the system and a significant amount of domain knowledge is required for being able to identify and manage it. It sometimes remains invisible to the developers and mainly affects the users and business departments. Domain debt tends to cross the borders of the system and its repayment sometimes also requires making organizational and cultural changes. It is typically caused by a lack of planning and domain analysis or organizational shortcomings in early stages of a project.

Based on the identified characteristics and effects we provide a checklist for being able to identify domain level TD in a system. This checklist is a first step towards being able to automatically detect and analyze domain level TD in the future. The checklist consists of 20 questions, divided into four weight categories. It asks questions about characteristics of domain level TD and if the score of the answers exceeds the weight threshold, we claim that the item of TD probably belongs to the category of domain level TD. The checklist has been validated by several senior software engineers.

In addition, this thesis analyzes the proportions and impacts of different levels of TD in a project from practice based on a list of TD items. This sanitized list of TD items is public and available for future research projects. Most of the TD debt in the list was dealing with structural and environmental issues. We were not able to prove statistical significance for results on the criticality, principal, and interest rates of the different levels of TD. Yet, we were able to prove that the interest payers and interest effects depend on levels of TD. The results confirm that interest on domain level TD is mainly

paid by the users and product owners of the system.

Domain level TD is a concept that illuminates the world of TD from a domain perspective and makes developers and other system stakeholders take into account that not all software problems are always only related to the technical quality of the system. In other words, developers should not just closely monitor the results from static code analysis tools, but the system should also be viewed in a higher and more abstract way. While implementation level TD allows developers to explain technical problems to non-technical stakeholders, domain level TD allows domain experts to explain domain-related problems to developers.

7. Future Work

This thesis takes a look at domain level TD at a smaller scale, involving a single company. In the future it would be interesting to conduct a more comprehensive and large-scale study on domain level TD, which looks at a number of different companies and organizations. This would allow to further validate the characteristics and effects of domain level TD in practice, presented in this thesis. The usefulness of the construct domain level TD should also be further investigated.

The domain debt identification checklist needs further work and validation in the future. It needs be tested in practice, by analyzing debt in different projects and in different contexts. Possibly, some questions have to be specified, removed, or added in the checklist. In addition, the categorization of the questions, their weighting, and the threshold are currently based on small-scale experimentation and need to be adjusted and validated through a larger study.

In general, more research would be needed in the future on higher level more abstract types of TD. It is possible that there are even more new useful types of TD that still need to be defined and characterized. To date, a lot of research has flown into analyzing and measuring implementation level TD, but for higher level TD, many of the questions still remain unanswered.

A. General Addenda

A.1. Case Study Form Template

Interviewee:
Version: Template V3

Case Study Template V3

Contextual information

System

1. Purpose *(What does the system do?)*
   - Main use cases

2. Domain *(What domain knowledge relevant to understand the debt?)*
   - Business context
   - Relevant domain knowledge
   - Relevant technical terms

3. Technologies *(Is the debt influenced by the technological landscape?)*
   - Legacy technologies *(e.g. Kobol)*

4. Architecture *(What does the high-level architecture of the system look like?)*
   - {microservices/monolith/cloud-based/...}

5. Size of the system *(What is the size of the system?)*
   - {small/medium/large}

Stakeholders

6. Count and distribution *(How many stakeholders and how are they distributed?)*

7. Field of the project *(What is the field of the project?)*
   - {defense/governmental/commercial/…}

8. Atmosphere *(How good is the atmosphere among the stakeholders?)*

Interviewee:
Version: Template V3

9. Clients *(Who are the clients of the project?)*
   - {BMW/Deutsche Telekom/...}

10. Users *(Who and how many?)*

Project

11. Structure *(Any limitations resulting from the project structure?)*
    - {outsourcing/on-site/...}

12. Development methodology *(What development methodology is/was used?)*
    - {agile/waterfall/...}

13. Age *(Since when has the system been developed?)*

14. Team size *(How many developers have been/are working on the system?)*

Identification and causes

Domain debt

15. Problem *(What is the domain debt about?)*

16. Why domain debt *(What makes it domain debt and not some other type of TD, e.g. architecture debt?)*

17. Location *(Where does the debt lie?)*
    - {architecture/data structure/code/...}

18. Nature of the problem *(Does it accumulate with time?)*
    - {contagious/stable}

Interviewee:
Version: Template V3

19. Observed state *(What did the observed state with the debt look like?)*
    - {wrong dependencies/incorrect names/...}

20. Target state *(What would the correct/optimal solution be?)*
    - {restructured modules/new technologies/...}

Causes

21. Causes of the debt *(What are the causes of the debt?)*
    - {business trade-offs/missing knowledge/time/budget/...}

22. Intent *(Was the debt intentional or strategic?)*
    - {deliberate/self-admitted/inadvertent/...}

Identification

23. Time *(When was the debt detected?)*

24. Activity *(How was the debt detected?)*
    - {due to the symptoms/refactoring phase/design phase/...}

25. Effort *(What effort was needed to completely identify and map the debt?)*

26. Domain knowledge *(Is domain knowledge needed to identify the debt and why?)*

Interviewee:
Version: Template V3

Effects and consequences

Consequences

27. Consequences of the debt *(What are/were the consequences of this debt?)*
    - {not being able to add new features/high maintenance costs/low efficiency/...}

28. Payers *(Which stakeholders felt the pain most?)*
    - {users/developers/sales department/...}

Cost

29. Principal *(What is the principal of the debt that is paid to fully eliminate the debt from the system?)*

30. Interest *(What is the interest of the debt that is paid at every time point "x" during the existence of the debt?)*

31. Interest probability *(What is the probability that the interest will have to be paid?)*

Criticality

32. Criticality of the debt *(How critical is/was the debt?)*
    - {low/medium/high}

Management

Management strategy

33. Decisions *(How has the debt been managed? What decisions have been made and why? How do they influence the debt?)*
    - {decided not to repay the debt/decided to use a workaround solution/decided to refactor/…}

Interviewee:
Version: Template V3

Current state

34. Lifecycle phase *(At what life-cycle phase is the debt currently?)*

- {still needs to be analyzed/identified/at work/...}

List of Figures

List of Tables

Bibliography

[1] W. Cunningham. "The WyCash Portfolio Management System". In: (1992). http://c2.com/doc/oopsla92.html, Accessed at 25.11.2020. OOPSLA 92 Experience Report.

[2] H. Störrle and M. Ciolkowski. "Stepping Away from the Lamppost: Domain-Level Technical Debt". In: (2019). 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). IEEE.

[3] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi. "Technical Debt: Towards a Crisper Definition". In: (2013). Report on the 4th International Workshop on Managing Technical Debt. ACM SIGSOFT.

[4] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton. "Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt". In: (2015). ESEC/FSE'15. ACM.

[5] B. Curtis, J. Sappidi, and A. Szynkarski. "Estimating the Size, Cost, and Types of Technical Debt". In: (2012). IEEE.

[6] E. Lim, N. Taksande, and C. Seaman. "A Balancing Act: What Software Practitioners Have to Say about Technical Debt". In: (2012). IEEE Software, IEEE Computer Society.

[7] H. Krasner. "The Cost of Poor Quality Software in the US: A 2018 Report". In: (2018). Consortium for IT Software Quality (CISQ). https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf, Accessed at 27.11.2020.

[8] "United Kingdom GDP". In: (2020). The World Bank Group. https://data.worldbank.org/country/GB, Accessed at 27.11.2020.

[9] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. "Managing Technical Debt in Software-Reliant Systems". In: (2010). ACM.

[10] N. S. Alves, T. S. Mendes, M. G. de Mendonça, R. O. Spínola, F. Shull, and C. Seaman. "Identification and Management of Technical Debt: A Systematic Mapping Study". In: (2015). Elsevier.

[11] T. Besker, A. Martini, and J. Bosch. "Technical Debt Cripples Software Developer Productivity". In: (2018). IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE.

[12] N. Rios, M. G. de Mendonça Neto, and R. O. Spínola. "A Tertiary Study on Technical Debt: Types, Management Strategies, Research Trends, and Base Information for Practitioners". In: (2018). Information and Software Technology 102. Elsevier.

[13] N. Zazworka, R. O. Spínola, A. Vetro, F. Shull, and C. Seaman. "A Case Study on Effectively Identifying Technical Debt". In: (2013). EASE'13. ACM.

[14] Dimensions. "Text "Technical Debt" in Title and Abstract". In: (2021). Dimensions. `https://app.dimensions.ai`, Accessed at 24.03.2021.

[15] "International Conference on Technical Debt (TechDebt)". In: (2020). ICSE 2019. `https://2019.techdebtconf.org/`, Accessed at 27.11.2020.

[16] M. M. Lehman. "Lehmans Laws". In: (1980). `https://wiki.c2.com/?LehmansLaws`, Accessed at 27.11.2020.

[17] S. Hacks, H. Höfert, J. Salentin, Y. C. Yeonga, and H. Lichter. "Towards the Definition of Enterprise Architecture Debts". In: (2019). 23rd International Enterprise Distributed Object Computing Workshop (EDOCW). IEEE.

[18] E. Allman. "Managing Technical Debt". In: (2012). Communications of the ACM.

[19] E. Tom, A. Aurum, and R. Vidgen. "An Exploration of Technical Debt". In: (2013). The Journal of Systems and Software 86. Elsevier.

[20] K. Schmid. "On the Limits of the Technical Debt Metaphor: Some Guidance on Going Beyond". In: (2013). Fourth Workshop on Managing Technical Debt, Workshop at the International Conference on Software Engineering, 2013. IEEE.

[21] E. da S. Maldonado and E. Shihab. "Detecting and Quantifying Different Types of Self-Admitted Technical Debt". In: (2015). IEEE.

[22] P. Kruchten, R. L. Nord, and I. Ozkaya. "Technical Debt: From Metaphor to Theory and Practice". In: (2012). IEEE Software, IEEE Computer Society.

[23] F. Shull. "Perfectionists in a World of Finite Resources". In: (2011). IEEE Software, IEEE Computer Society.

[24] A. Chatzigeorgiou, A. Ampatzoglou, A. Ampatzoglou, and T. Amanatidis. "Estimating the Breaking Point for Technical Debt". In: (2015). IEEE.

[25] P. Avgeriou, P. Kruchten, I. Ozkaya, C. Seaman, and R. Nord. "Managing Technical Debt in Software Engineering". In: (2016). Dagstuhl Seminar 16162. Dagstuhl Reports, Vol. 6, Issue 4, pp. 110–138.

[26] M. Fowler. "Technical Debt Quadrant". In: (2009). `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, Accessed at 25.11.2020.

[27] O. Lachish. "Introduction to Software Engineering: Tools and Environments". In: (2012). `http://www.dcs.bbk.ac.uk/~oded/OODP12/Sessions/Session-10-rev-1.pdf`, Accessed at 27.11.2020.

[28] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola. "Towards an Ontology of Terms on Technical Debt". In: (2014). 6th IEEE International Workshop on Managing Technical Debt. IEEE.

[29] Z. Li, P. Avgeriou, and P. Liang. "A Systematic Mapping Study on Technical Debt and its Management". In: (2015). The Journal of Systems and Software 101. Elsevier.

[30] D. A. Tamburri and E. D. Nitto. "When Software Architecting Leads to Social Debt". In: (2015). 12th Working IEEE/IFIP Conference on Software Architecture. IEEE.

[31] Z. Codabux and B. Williams. "Managing Technical Debt: An Industrial Case Study". In: (2013). IEEE.

[32] P. Alexander, S. Hacks, J. Jung, H. Lichter, U. Steffens, and Ö. Uludağ. "A Framework for Managing Enterprise Architecture Debts - Outline and Research Directions". In: (2020). 10th International Workshop on Enterprise Modeling and Information Systems Architectures.

[33] D. Thomas and A. Hunt. *The Pragmatic Programmer*. Pearson Education Inc. 2019.

[34] M. E. Conway. "Gesetz von Conway". In: (1968). t2informatik GmbH. `https://t2informatik.de/wissen-kompakt/gesetz-von-conway/`, Accessed at 27.11.2020.

[35] C. Criddle and L. Kelion. "Coronavirus Contact-Tracing: World Split Between Two Types of App". In: (2020). BBC News. `https://www.bbc.com/news/technology-52355028`, Accessed at 27.11.2020.

[36] "In Diesen Ländern Stecken Sich die Meisten Deutschen Reisenden An". In: (2020). WELT News. `https://www.welt.de/vermischtes/article213491558/RKI-Statistik-zu-Corona-In-diesen-Laendern-stecken-sich-die-meisten-Deutschen-an.html`, Accessed at 27.11.2020.

[37] N. Rios, R. O. Spínola, and M. G. de Mendonça Neto. "Supporting Analysis of Technical Debt Causes and Effects with Cross-Company Probabilistic Cause-Effect Diagrams". In: (2019). IEEE/ACM International Conference on Technical Debt (TechDebt). IEEE.

[38] N. Davis. "Driving Quality Improvement and Reducing Technical Debt with the Definition of Done". In: (2013). Agile Conference. IEEE.

[39] H. McCloskey. "The Definition of Done: What Product Managers Need to Know". In: (2020). ProductPlan. `https://www.productplan.com/agile-definition-of-done/`, Accessed at 27.11.2020.

[40] P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering: Guidelines and Examples*. A John Wiley & Sons, Inc., Publication. 2012.

[41] I. Campbell. "Chi-Squared and Fisher–Irwin Tests of Two-by-Two Tables with Small Sample Recommendations". In: (2007). Statist. Med., 26: 3661-3675. John Wiley & Sons, Ltd.

[42] A. Patrizio. “What is Multi-Tenant Architecture?” In: (2021). Datamation. https://www.datamation.com/cloud/what-is-multi-tenant-architecture/, Accessed at 26.02.2021.

[43] L. H. A. Service. “How Many Parts are on a Car?” In: (2020). Lee Hill Auto Service. https://leehillautoservice.net/blog/view/how-many-parts-are-on-a-car, Accessed at 26.02.2021.

[44] J. Schulz. “Why Column Stores?” In: (2018). Pythian. https://blog.pythian.com/why-column-stores/, Accessed at 26.02.2021.

[45] L. Downey. “Golden Hammer”. In: (2020). Investopedia. https://www.investopedia.com/terms/g/golden-hammer.asp, Accessed at 26.02.2021.

[46] J. Brustein and N. Sanders. “The Real Story of How Amazon Built the Echo”. In: (2016). Bloomberg L.P. https://www.bloomberg.com/features/2016-amazon-echo/, Accessed at 27.11.2020.

[47] B. Wolford. “What is GDPR, the EU’s New Data Protection Law?” In: (2020). GDPR.EU. https://gdpr.eu/what-is-gdpr/, Accessed at 27.11.2020.

[48] N. Statt. “Amazon Sent 1,700 Alexa Voice Recordings to the Wrong User Following Data Request”. In: (2018). VOX Media, The Verge. https://www.theverge.com/2018/12/20/18150531/amazon-alexa-voice-recordings-wrong-user-gdpr-privacy-ai, Accessed at 27.11.2020.

[49] L. Kelion. “Amazon Alexa: Luxembourg Watchdog in Discussions About Recordings”. In: (2019). BBC News. https://www.bbc.com/news/technology-49252503, Accessed at 27.11.2020.

[50] M. Day, G. Turner, and N. Drozdiak. “Amazon Workers Are Listening to What You Tell Alexa”. In: (2019). Bloomberg L.P. https://www.bloomberg.com/news/articles/2019-04-10/is-anyone-listening-to-you-on-alexa-a-global-team-reviews-audio, Accessed at 27.11.2020.

[51] D. staff (sms). “Telekom Privacy Scandals Multiply Amid Calls for Data Protection”. In: (2008). Deutsche Welle (DW). https://www.dw.com/en/telekom-privacy-scandals-multiply-amid-calls-for-data-protection/a-3692894, Accessed at 27.11.2020.

[52] M. Weidenbrück. “Hello Magenta! With Smart Speaker, Your Home Listens to Your Command”. In: (2017). Deutsche Telekom AG. https://www.telekom.com/en/media/media-information/consumer-products/with-smart-speaker-your-home-listens-to-your-command-508276, Accessed at 27.11.2020.

[53] Techopedia. “Ground Truth”. In: (2021). Techopedia. https://www.techopedia.com/definition/32514/ground-truth, Accessed at 26.02.2021.

[54] J. Palm. “Masterarbeit Joonas Palm - Sanitized List of Technical Debt Items”. In: (2021). sebis TU München. https://wwwmatthes.in.tum.de/pages/14sr6fu61bled/Masterarbeit-Joonas-Palm, Accessed at 24.03.2021.

[55] Artifacts. “What Is an Artifact? Everything You Need to Know”. In: (2020). Artifacts. https://artifacts.ai/what-is-an-artifact/, Accessed at 12.03.2021.