Aalto University
School of Science
Master's Programme in ICT Innovation

Zsolt Dargó

Technical Debt Management:

Definition of a Technical Debt Reduction Software Engineering Methodology for SMEs

Master's Thesis
Espoo, July 25, 2019

Supervisors: Professor Petri Vuorimaa, Aalto University
Professor Carlos Ángel Iglesias Fernández, Polytechnic University of Madrid
Advisor: David Muñoz Díaz M.Sc. (Tech.)

Aalto University
School of Science
Master's Programme in ICT Innovation

ABSTRACT OF
MASTER'S THESIS

Author: Zsolt Dargó

Title:
Technical Debt Management: Definition of a Technical Debt Reduction Software Engineering Methodology for SMEs

Date: July 25, 2019 **Pages:** vi + 89

Major: Software and Service Architectures **Code:** SCI3082

Supervisors: Professor Petri Vuorimaa
Professor Carlos Ángel Iglesias Fernández (Polytechnic University of Madrid)

Advisor: David Muñoz Díaz M.Sc. (Tech.)

In the past two decades, the metaphor of technical debt has gained significant importance in the field of software engineering. In general, the term is used to describe scenarios when instead of providing a proper solution for a given task, a sub-optimal implementation is used in order to gain short term benefits. Unfortunately, this kind of decisions can - and most of the time do - result in increased maintenance costs and poor evolvability in the long run. Over time, software practitioners further refined the initially source code-focused concept and started to apply the metaphor for a much wider range of software engineering inefficiencies, such as architectural defects, inappropriate documentation or low test coverage. Due to its similarity to financial debt, the analogy has also become a valuable communication tool in situations when there are less technical people involved in discussions.

This master's thesis defines a technical debt reduction methodology, which can help SMEs to control the accumulation of technical debt. The proposed methodology can be thought of as a set of steps and good practices that facilitate the long-lasting productivity and profitability of SMEs. Since this field of research is relatively new, the need for publications addressing the topic is still rather high.

Due to its numerous negative effects, it is crucial for companies to keep their debt levels as low as possible, which requires a systematic way of managing technical debt. Besides providing such a methodology, the document also intends to raise awareness about the nature and dangers of taking on unreasonable amounts of debt by examining the most important characteristics of the phenomenon. Finally, the thesis presents an industrial case study as well, which aims to showcase how some of the most necessary steps can be taken in practice.

Keywords: software engineering, technical debt, technical debt management, methodology

Language: English

Acknowledgements

I would like to express my gratitude towards both of my academic supervisors first. To start with, I really appreciate the continuous support provided by prof. Petri Vuorimaa. He made it sure that my master's thesis could also comply with the requirements of Aalto University, even though I did my internship, completed the project and wrote the document in Spain.

I am also thankful to prof. Carlos Ángel Iglesias Fernández for always helping me by steering my master's thesis project in the right direction. He was always ready to help me and shared very valuable feedback about my work on multiple occasions.

Likewise, I would not have been able to complete my project without the contributions of my supervisor at the case study company, David Muñoz Díaz M.Sc. I would like to thank him for helping me define the project itself, for giving my project the highest priority at all times, for sharing some extremely useful insights and for aiding my learning process in general.

Furthermore, I am also grateful to EIT Digital Master School and my two universities involved in the program, Aalto University (Finland) and Polytechnic University of Madrid (Spain). I would not have had the chance to write about such an interesting topic after two years of studying abroad, if it was not for the double-degree program organized by them.

Lastly, I truly appreciate all the support provided by my family, friends, colleagues and everybody who encouraged me in some way during the writing process. Thank you everyone!

Espoo, July 25, 2019

Zsolt Dargó

Contents

Chapter 1

Introduction

This chapter serves as an overview of the master's thesis. It provides the problem statement, enumerates the main goals and also describes the structure of the document.

1.1 Problem statement

The problem of technical debt affects a large number of small and medium-sized enterprises (SMEs) in the field of software development. This term refers to those "quick and dirty" solutions (e.g., shortcuts, workarounds) that are implemented to gain short term benefits in exchange for productivity loss in the long term. Furthermore, this type of debt can also easily get out of control, resulting in a so-called debt spiral. As the survey of Ernst et al. [12] also indicated, most IT practitioners do not really understand the real weight of the issue and they generally lack awareness about the topic as well. As a direct consequence, many companies just silently suffer from the increasing number of negative effects caused by the phenomenon, since they do not know how to escape from their troublesome situations.

This kind of scenarios usually arise in companies because they tend to surrender to technical debt way too early, thinking that it is already too late to rectify their current situations. However, they should make an effort to stop the spiral as soon as possible, before it truly becomes uncontrollable. The main point is that the management of technical debt can be started at any point in time and even small changes can make a considerable difference. Given the importance of the issue and the high level of complexity of the already existing approaches, the core of this master's thesis is a user-friendly technical debt reduction methodology that can aid software development SMEs in keeping their technical debt at a satisfactory level.

This master's thesis has three main goals. These can be found in the list below in a logical order:

1. **Spread awareness:** As mentioned earlier, one of the primary problems is that people in general do not know enough about the topic in order to truly understand the implications of technical debt. Hence, this document tries to collect and summarize information about related aspects in a systematic way.

2. **Define the methodology:** Although this is the principal goal of the document, it only occupies second place in this list, since it greatly builds upon the success of the first goal. As stated earlier, it is intended to be easy to use and cover every necessary aspect of technical debt management.

3. **Describe the case study:** The document also describes an industrial case study to showcase the usefulness and benefits of the methodology. It provides details about the practical usage of every step and shares my most significant achievements as well.

1.2 Structure of the Thesis

The remaining of this document is structured as follows:

Chapter 2 introduces the state of the art and serves as a thorough literature review. To start with, it provides an overview about technical debt. Next, it introduces the relevant properties, categories, sources and effects of technical debt. Finally, it also discusses the topic of technical debt management.

Chapter 3 defines the technical debt reduction methodology. After explaining the overall concept, it contains details about all the 7 plus 1 steps in logical order: understanding the environment, identification, measurement, monitoring, prioritization, repayment communication and preventive steps.

Chapter 4 contains the case study. Logically, it starts by introducing the environment where it was carried out, followed by the discussion of the implementation and the results of every step.

Chapter 5 evaluates the outcomes of the master's thesis project.

Chapter 6 reflects upon the work as a whole, draws conclusions and also addresses the topic of future work.

Chapter 2

State of the Art

This chapter provides a literature review about technical debt. It not only defines the metaphor itself, but it also describes various aspects related to it.

2.1 Technical debt overview

Ever since the notion of technical debt was born, there have been many different approaches to define technical debt and explain the corresponding concepts and terminologies. This section aims to address this issue by establishing a common understanding of the most important technical debt-related definitions and terminologies. In order to do that, the following subsections present some of the most recurring ideas of the already existing scientific literature.

2.1.1 What is technical debt?

The technical debt metaphor was introduced by Ward Cunningham [9] for the very first time at one of the OOPSLA (*Object-Oriented Programming, Systems, Languages & Applications*) research conferences in 1992. He mentioned this analogy with the goal of explaining the trade-off between the fast delivery of low-quality software code, and thus resulting high maintenance costs. His definition of technical debt was the following:

> *"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. Entire engineering organizations can be brought to a stand-still under the debt*

> *load of an unconsolidated implementation, object-oriented or otherwise."*

However, this is certainly not the only definition that has ever been published. During the past two decades, many other scientific authors and software engineering experts — especially members of the agile community — refined and broadened the original metaphor. According to the research of Kruchten et al. [21], as time passed by, the concept was diluted and applied for several other phenomena of the software development project life cycle as well. In their opinion, IT professionals started to overuse the term for essentially any type of issues that undermined the success of software development projects. Thus, as a direct consequence, the analogy between monetary debt and technical debt also lost some of its strength.

Furthermore, Fowler [14] also addressed the topic of technical debt, further elaborating on the original concept. In his definition, he moved the focus away from the source code and talked about features of a system in a more generic way:

> *"You have a piece of functionality that you need to add to your system. You see two ways to do it, one is quick to do but is messy - you are sure that it will make further changes harder in the future. The other results in a cleaner design, but will take longer to put in place."*

Fowler [14] also highlighted the importance of deciding whether creating technical debt in a given situation is necessary or not. According to him, both approaches can have reasonable explanations and technical debt is neither inherently good nor bad. On the one hand, taking on technical debt is sometimes a good idea: just like when a business borrows some capital to benefit from a market opportunity, going into technical debt can help with making progress and delivering features faster. For instance, Allman [1] gives the example of using a slow, but simple algorithm in a prototype where a much faster one will be needed instead in the production environment. This decision creates technical debt, but it is completely acceptable as long as it is tracked and developers know for a fact that a better algorithm exists and can be implemented for the same task later on. However, on the other hand, they both acknowledged that technical debt can become crippling in the long run and even though it can provide an initial boost, it usually involves sacrificing some – or a significant portion – of the future development progress.

Lim et al. [26] carried out an interview study, in which they found some rather interesting facts about the practical use and common understanding

of the metaphor. The study was designed to determine how IT professionals perceived technical debt in their everyday work lives. Although definitions given by IT practitioners agreed on the compromise between "expedient short-term decisions" and "long-term costs", the focus of the whole technical debt concept was often different. As reported by them, technical debt can equally be artifact-oriented (including, but not limited to source code) or task-oriented (considering actions that should have been done in the past). Either way, definitions always place a significant emphasis on some kind of "trade-off among quality, time, and cost".

In the same paper, Lim et al. [26] also mentioned that technical debt is perceived in two significantly different ways by software engineers and management people. While the former group considers technical debt as a state that needs to be avoided by all means, the latter group embraces its existence and thinks of it as yet another strategic tool. The reason behind this difference in attitude is simple. On the one hand, programmers are the ones who do the actual technical work, therefore, they experience the potentially negative effects directly in their everyday work and they prefer to create "perfect software". On the other hand, management people are much more used to taking risks and meeting deadlines, since those are things that business life demands anyway. Consequently, they understand that there are times when the only way of making progress is borrowing some work effort from the future.

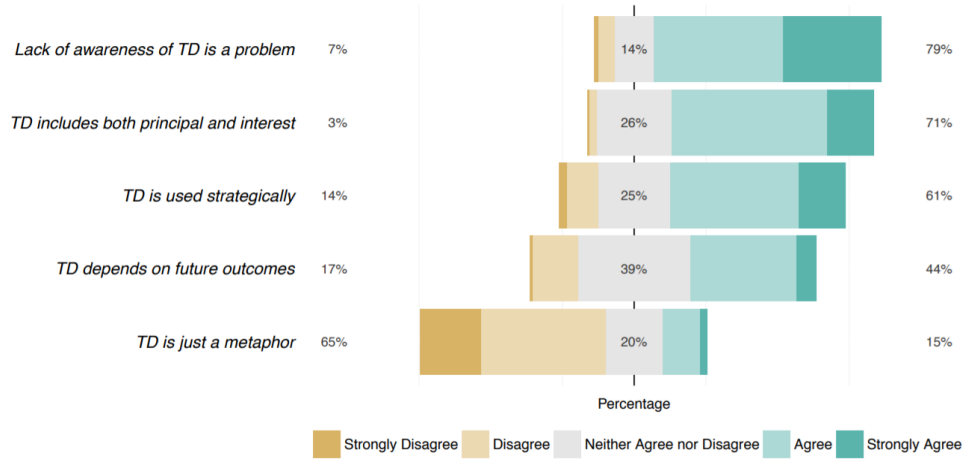

Figure 2.1: Statistics of high-level technical debt definitions by Ernst et al. [12].

Ernst et al. [12] also addressed the topic of technical debt definitions

given by software practitioners in a survey-based paper, published in 2015. As it can be seen from their visualization above (Figure 2.1), participants of the survey mostly agreed on high-level aspects of the metaphor. The three percentages in each line stand for the proportions of SD + D, N, A + SA answers, respectively.

One of the most compelling conclusions of the statistics is that software practitioners think that awareness should be raised about the importance of technical debt. This is in consonance with the findings of Lim et al. [26], since they found that 75% of the interview participants were new to the technical debt term. Additionally, participants of the survey agreed that technical debt includes principal and interest as well (which will be further described later). Lastly, technical debt is perceived as a strategical tool and it is dependent on future outcomes.

2.1.2 What are the main attributes of technical debt?

Just like in case of financial debt, a certain amount of interest is incurred in the majority of cases, in accordance with what was stated by the original metaphor as well [9]. By creating technical debt, organizations also create monetary debt for themselves and projects can actually go bankrupt due to the technical debt they accumulate. Therefore, when addressing the topic of technical debt, we can distinguish five main attributes, which are described below:

- **Principal:** In consonance with monetary terms, this part of a debt refers to the originally borrowed amount of money. Analogously, the principal part of technical debt represents the work that is not done well right away, but it is substituted by a workaround, a shortcut or simply a poor solution with the goal of accelerating development.

- **Interest:** In our everyday lives, this portion of a debt can be thought of as the cost of borrowing money. The price we have to pay usually depends on the the principal, because interest is normally defined as a percentage of that. In a very similar fashion, technical debt also comes with a price in the form of extra work that has to be carried out first, in order to "pay off" the principal part of the debt. In practice, this usually means the re-implementation of features that were created after the debt itself had been introduced, since they most likely built upon several characteristics of the sub-optimal solution that has to be changed when the debt is paid off.

- **Interest probability:** In line with the description of Guo et al. [17],

this attribute is associated with the probability of having to actually pay an interest. This piece of information can prove to be rather advantageous in making management decisions.

- **Monetary cost:** At the end of the day, technical debt boils down to actual monetary debt as Tom et al. [39] also indicated in their article. Every time a developer has to fix blocking factors (e.g., bugs, regressions, lack of necessary abstraction layers) before starting the implementation of an actual feature, his or her time is wasted. This directly translates to increased monetary costs for the organization, since the time of developers is expensive.

- **Bankruptcy:** It is hard to define a point in time when one can declare that a project went bankrupt due to technical debt, since this kind of debt cannot be easily measured in an objective manner. However, as Hilton [18] described it in one of his blog posts, the definition of the state of bankruptcy could be put the following way: when an organization can no longer do feature development and have to either pay down all the technical debt at once or completely rewrite the software, the project has reached the status of bankruptcy.

Figure 2.2 illustrates some characteristics of technical debt. As it can be seen, component 1 and 2 were not constructed properly, which resulted in a hole in the overall structure (technical debt). Later on, as time passed by, components 3–6 were built upon the previous two. If the need of fixing component 1 and 2 arises at some point during the life cycle of the project, not only the two incomplete components need to be replaced (paying principal), but it also requires deconstructing and rebuilding components 3–6 (paying interest). However, a related point to consider is that technical debt can be left unpaid in some of the situations, when its existence does not pose a real threat to the usability, maintainability and success of a given system.

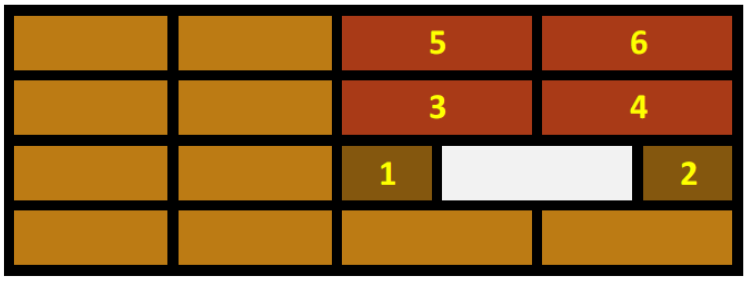

Figure 2.2: Illustration of technical debt-related principal and interest

2.1.3 When and how is technical debt paid back?

By default, the word "debt" implies the necessity of repayment. As already mentioned, the most noteworthy difference between monetary and technical debt is that the latter one is not necessarily paid back on any fixed schedule. Furthermore, it is close to impossible to pay back technical debt completely, while that is an obligation when it comes to monetary debt. In everyday life, debts are usually paid back with periodical (e.g., monthly) payments, including interests. However, technical debt interest is paid every time a person is delayed in their work, because of the lower quality of the components involved that have technical debt.

Interestingly, the schedule is not the only difference here. While real life financial debts usually have to be paid back by the person that takes them on, technical debt is often paid off by other members of a given organization. Furthermore, the person who originally created the debt, might not even work for the organization anymore at the time of repayment. This means that many times there is no real incentive for people working for an organization to avoid technical debt.

2.1.4 Why is technical debt dangerous?

The term "debt spiral" (sometimes referred to as "vicious circle of debt") is very well-known in the world of finance. It stands for the phenomenon when paying back an existing debt simply forces the person in debt to borrow more and more money, thus, increasing their overall debt due to interest that they have to pay. Why is it called a spiral? Because it is virtually never ending cycle that is extremely hard to stop once it gets out of control.

Unfortunately, the world of technical debt is not free of debt spirals either and they can cause just as many issues as in case of monetary debts. Therefore, (technical) debt management (discussed further in section 2.6 of the chapter) has a crucial role in everyday life and software development as well.

2.2 Properties of technical debt

Discussions about technical debt usually involve various properties of it. Hence, it is essential to understand what aspects of the phenomenon are relevant to them, especially with respect to categorizing, prioritizing and managing technical debt in general. Some of the most relevant ones discussed by Brown et al. [7] and Ramakrishnan [35] can be found below:

Property name	Description
Visibility [7]	Technical debt is often invisible, which can cause some serious problems and surprising situations. For instance, if a developer makes a shortcut in order to meet an important deadline, but does not make the created technical debt visible to others, his or her colleagues can possibly face some difficulties when trying to do their own tasks. They would assume that things are properly implemented (according to known best practices) and try to build on top of the work done by the developer who introduced invisible technical debt and they would most likely find unexpected obstacles.
Value [7]	In general, well-managed debt can be a strategic tool to create value. For example, as the real life example of Brown et al. [7] stated it, having a mortgage makes it possible to buy a house even if we would not have enough money to pay for the whole building without borrowed money. They also added that the value can be thought of as the difference between the current state of something and a desired, more ideal state of it. Translating this into the field of software engineering, for instance skipping the creation of tests before a deadline, creates the value of saying that "the task was completed on time" (at least the customer facing part of it). However, this also means converting test writing into technical debt, since it still should be done at some point in the future.
Environment [7]	Every software engineering project has a different context. Therefore, it is of no surprise that technical debt is also relative to the context of the project in question. In other words, perfectly good implementation details belonging to one environment can become technical debt in other, but otherwise similar ones.

Origin of debt [7]	Technical debt can be created in many ways in software development. Identifying its source can be beneficial when it comes to debt management and prioritization. Brown et al. [7] distinguished strategically and unintentionally created technical debt as an example, however, this thesis will also introduce possible origins in a later section.
Impact of debt [7]	This property is related to the scope of technical debt. While some types of technical debt have a more localized impact, others can have an influence of a much wider range, effecting entire systems. Clearly, when prioritizing pieces of technical debt, those pieces have higher priority that belong to the latter category.
Longevity of debt [35]	Ramakrishnan [35] discussed another, rather important property of technical debt, which has to do with its intended duration. In his article, he talked about *short-term debt*, which should be paid off as soon as possible (e.g., in the next release cycle) and *long-term debt*, which can be left unpaid for even a few years. While the former type is intended for tactical measures, the latter type is more proactive and strategic.

2.3 Technical debt categorization

Organizing and categorizing software development-related issues are beneficial when it comes to solving them, since self-evidently, different categories of debt also require different approaches in mitigating them. For this reason, this subsection presents two perspectives of classifying technical debt. After the discussion of Fowler's technical debt quadrants, it also provides a set of a more specific categories that can be used during the identification, reduction and avoidance of technical debt.

2.3.1 The technical debt quadrants

Martin Fowler [15] not only addressed the relevance of the technical debt metaphor, but also discussed the topic of technical debt categorization. In his opinion, technical debt can fall into one of the four categories depicted

by his quadrants below. Unlike Robert C. Martin [37], he did not think that "messy code" was to be excluded from technical debt. According to Fowler, the real question was whether the metaphor was able to help in dealing with issues and communicating them to less technical people. Reaching the conclusion that the analogy was powerful enough to determine which technical inefficiencies are acceptable and which are not, he introduced the following four categories: *reckless–deliberate, reckless–inadvertent, prudent–deliberate and prudent–inadvertent.*

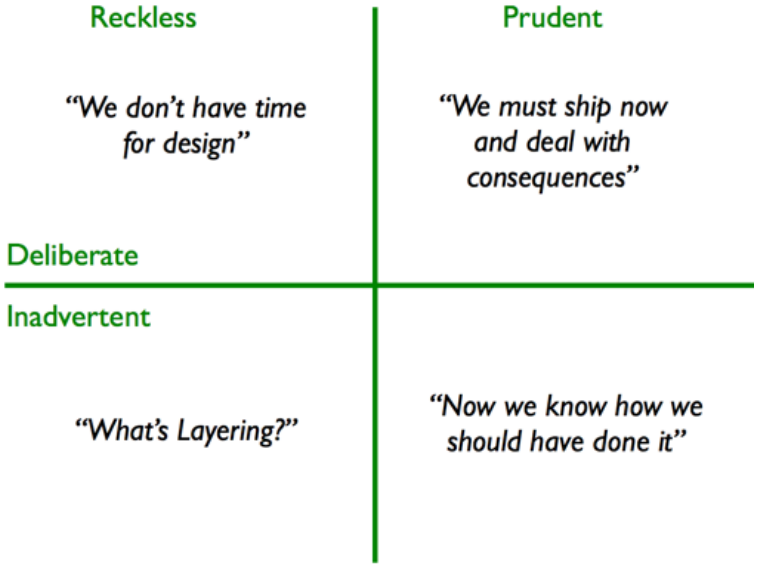

Figure 2.3: The technical debt quadrants proposed by Fowler [15]

In Fowler's opinion, *"the useful distinction isn't between debt or non-debt, but between prudent and reckless debt"*. Following that logic, unprofessionally and carelessly written code also counts as technical debt and falls into the reckless half of the figure, in contrast with what Uncle Bob stated [37].

The four segments can be explained more in detail the following way:

- **Reckless–deliberate debt:** This kind of debt is born due to the carelessness of developers. In some cases, even though they know that they are creating debt, they do not realize (or want to realize) what possibly crippling effects it might have. This type of attitude could be explained by the already mentioned phenomenon: developers do not have much incentive to avoid creating debt, as they might not even work at the company anymore if and when the debt has to be paid back. As Fowler pointed it out in his blog post, this does not necessarily have anything to do with the lack of knowledge, skills of developers or awareness of

design principles. These decisions are usually made based on the "budget" of a given project; in a business-driven development environment as Yli-Huumo [41] stated in his PhD thesis. The quote appearing in the corresponding quadrant represents a very typical managerial sentence when reckless–deliberate debt is accrued.

- **Reckless–inadvertent debt:** Reckless debt can be created inadvertently as well. In this case, there is a lack of professional knowledge and awareness of software engineering best practices. In a way, this might be the most dangerous type of debt, since the development team is not aware of its existence, and thus, it can lead to very unexpected, negative surprises during the development process. The only way of avoiding this is employing developers who are competent enough at a given job and make the minimum amount of mistakes. However, as a key takeaway of the interview series that Lim et al. [26] carried out, they found that most of the technical debt was born out of conscious decisions. As a matter of fact, only around one-fourth of the cases involved the sloppiness of employees.

- **Prudent–deliberate debt:** This category includes debt which is created on purpose, in order to reach a short-term goal. An important thing to mention here is that in this case, potential long-term consequences are taken into consideration and are thoroughly evaluated and this is what the quadrant quote demonstrates: "We must ship now and deal with consequences". Therefore, similarly to the other deliberate category and in consonance with Ramakrishnan [35], this type of debt also serves as a strategic tool to achieve business goals with compelling ROI ratios.

- **Prudent–inadvertent debt:** As the last category, prudent and at the same time inadvertent debt is a bit harder to imagine at first. But the key aspect here is that this class of debt can only be identified in retrospect as a result of evaluating the existing solution and reflecting on the entire learning process, which usually characterizes any software development project. According to Fowler [15], *"The point is that while you're programming, you are learning. It's often the case that it can take a year of programming on a project before you understand what the best design approach should have been."*. He also made a reference to a concept introduced by Brooks Frederick [6]. In their book, they proposed building "throw-away" projects first just for learning purposes simply, before implementing an actual solution.

2.3.2 Specific types of technical debt

The field of technical debt management is still a very recent area of research, which means that creating ways of organizing information into data and knowledge is very much needed. Alves et al. [3] identified the need for creating a technical debt ontology and define technical debt types and wrote about the topic in 2014. Later on, they revisited the topic in 2016 [2] and among other things, further refined the set of technical debt types. However, they were not the only ones who made a contribution; Brown et al. [7], Morgenthaler et al. [30], Bohnet & Döllner [4], Tom et al. [39], Ernst [11] and Greening [16] also addressed the question. Some of the most relevant technical debt types and their characteristics can be found at the end of this document, in Appendix A.

2.4 Sources of technical debt

This section discusses the topic of technical debt sources. First, it addresses the high-level origin of technical debt. Secondly, it introduces the technical debt landscape created by Kruchten et al. [21]. Thirdly, it lists the most relevant sources and their characteristics. Finally, it presents a ranking of more specific sources, created by IT practitioners.

According to Holvitie et al. [19], as a high-level way of categorization, technical debt sources can be grouped into four groups:

- Legacy from an earlier team working on the same project/product (57%).
- Legacy from an unrelated project/product within the organization (11%).
- Legacy from outside the organization, for instance as a result of an acquisition (7%).
- Non-legacy sources (25%).

2.4.1 The technical debt landscape

In 2012, Kruchten et al. [21] published an article that addressed the technical debt metaphor with the goal of transitioning the metaphor to theory and practice. In their article, they described a landscape of technical debt, which aims to organize types of debt based on their sources and visibility. The landscape can be found in Figure 2.4.

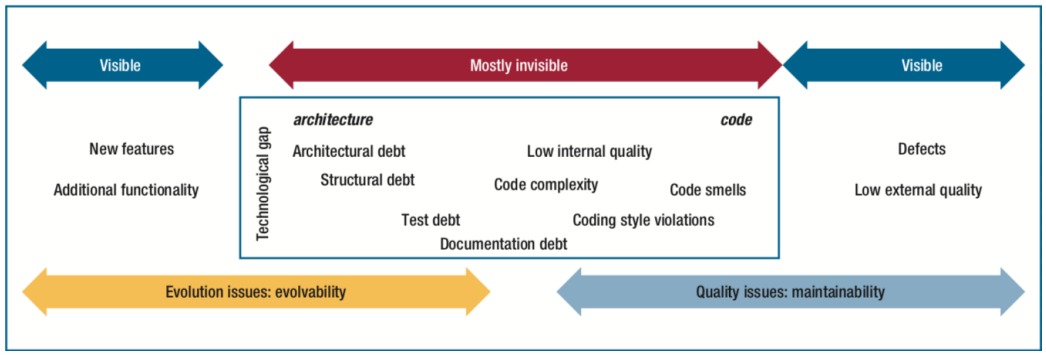

Figure 2.4: The technical debt landscape by Kruchten et al. [21]

It is important to note that according to the creator of the landscape, the metaphor should only cover invisible issues belonging to the inner part of the picture, having a blue border. In this case, visibility is considered from the point of view of clients and not that of software developers. As explained later on, including the rest of the problems would excessively dilute the metaphor.

The landscape organizes debt sources into two groups: those types of debt that are related to the ease of adapting to changes (*evolvability*) are grouped on the left, while aspects of keeping the product properly functioning and serving its intended purpose (*maintainability*) can be found on the right. It is also noteworthy that the "technological gap" expression on the left refers to poor choices of technology, since choosing the wrong technology for a given purpose in the present can equally generate obstacles in the future. Unfortunately, some of the software practitioners tend to focus on the right side of this landscape exclusively, which can lead to serious complications in the long run.

One year after he published his first paper, Kruchten et al. [20] revisited the topic of the technical debt landscape, presumably because he noticed that the confusion about the metaphor and its usage was growing. As they explained, including some phenomena of software development as source of technical debt might excessively dilute the metaphor, thus making it lose some of its utility and power. Some of the most prevalent misconceptions that he identified are:

- limiting possible sources of technical debt to *bad source code quality*,
- counting *defects* as technical debt,
- considering *not yet implemented new features* as technical debt.

First, with respect to bad code quality, he declared that technical debt is not only about source code (as it can be seen in later parts of this document).

It is not a surprise that program code is the first thing that comes to mind, when technical debt is mentioned, since its issues are the easiest to be identified, owing to the existence of numerous static code analysis tools. Even though this makes them more visible to developers, there are countless other ways of how debt can be accrued during development. One good example is the kind of debt that is usually linked to the structure, architecture or the set of technologies that are used by a given system.

Secondly, as per defects and bugs, he emphasized that they belonged to external qualities of the code and the metaphor was only intended for the internal ones. Furthermore, he also underlined another important fact: technical debt exerts its effects only in the future. Therefore, this serves as another reason why defects should not be treated as source of debt, since their effect can be seen in the present already. In addition, defects are visible not only to the developer team, but also to the clients of an organization, which serves as another argument why defects should not be considered technical debt.

Finally, it is also important to note that while not yet implemented, new features do not create technical debt, mistakes such as insufficient requirement analysis, poor requirement prioritization and misunderstanding of requirements can lead to requirement debt.

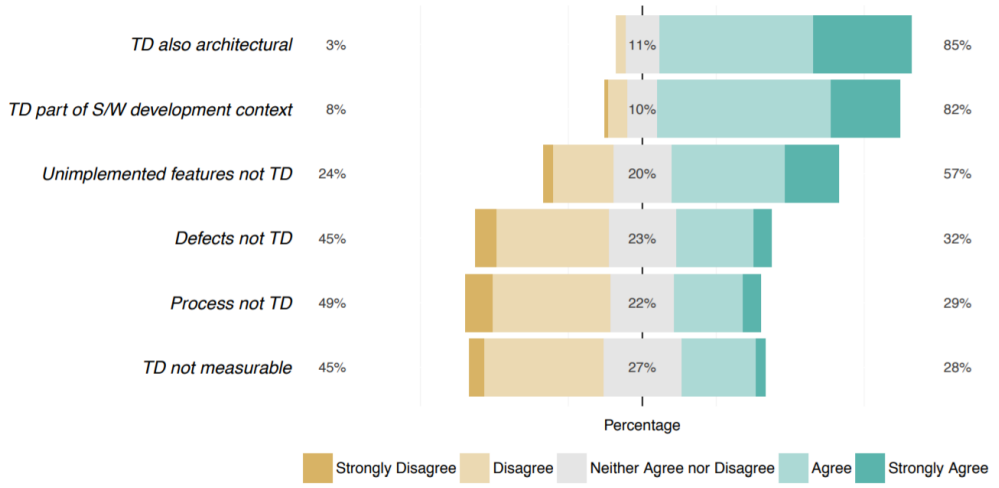

Figure 2.5: Typical sources of technical debt by Ernst et al [12]

On a related note, findings and statistics of Ernst et al. [12] seem to support Kruchten's aforementioned views about common misconceptions. Based on the statistical data presented in Figure 2.5, 85% of the respondents agreed or strongly agreed that technical debt is also architectural, therefore,

it should not be limited to code quality. Additionally, only 24% of the interviewees considered unimplemented features as technical debt which also supports Kruchten's opinions. However, the question whether defects should be counted as technical debt or not proved to be a divisive topic, since 45% considered defects as technical debt and 32% did not.

2.4.2 Most significant technical debt sources

The following subsections present the most relevant technical debt sources. The information about each of them is supported by other authors who have also done extensive research on the topic. Much of the literature cited below was written based on surveys, which means that they represent debt sources that were identified in real life projects. In this section, the intention of the author was to present main debt sources in a somewhat unpolished order with respect to the importance, impact and prevalence of each of them. A more detailed ranking of debt source types will be presented in the next section.

Pragmatism

The first two technical debt sources — pragmatism and prioritization — go hand in hand. However, they are discussed separately in this thesis, just like Tom et al. [39] did it in their publication. In consonance with the meaning of the word "pragmatism", debt can be created as a result of practical considerations.

In order to explain the importance of practical decisions, Brazier [5] gave the example of a small company in a niche market. According to his example, their only chance of succeeding as a company is entering the market with their product as first, so that they are visible to potential customers as early as possible. Otherwise, competition can easily take away the business opportunity. As pointed out by Brazier [5], considering the long-term effects of short-term decisions is not very sensible in this kind of situations, since the "long term" does not even exist without the short-term success of the organization.

Lim et al. [26] also discussed how pragmatism can result in technical debt. They also confirmed that there are good marketing opportunities or shopping windows, when taking on debt has to be done with the goal of not wasting those favorable circumstances. As two examples of such opportunities they mentioned are the possibility of acquiring funding and obtaining early customer feedback to better adjust features of a product according to the real needs of customers. All the arguments of Lim et al. [26] emphasize

that technical debt is a balancing act between software quality and business reality, due to competing concerns.

Prioritization

Prioritization is often born out of pragmatism and it can also result in deliberate technical debt. Tom et al. [39] suggested that prioritization requires trade-offs in non-functional aspects of a product, which creates into technical debt.

Lim et al. [26] indicated that time constraints introduced by tight deadlines, contractual obligations or the integration with a partner product can force managers to take prioritizing measures, such as decreasing the time spent on not only code and design reviews, but also on the creation of unit tests for instance. This is how companies end up with only "satisfactory" features as the result of a deliberate prioritizing decision. These features still require some work in the future, which by definition equals to technical debt and it should be avoided in the vast majority of the cases.

This phenomenon also has to do with expectation management as Hilton [18] stated it. Customers of a product can easily get used to an average pace of feature delivery. When for some reason the natural pace of a development team slows down, they still expect the pace that they got used to and living up to those expectations can only happen by cutting corners.

Allman [1] mentioned the widely used project management triangle (see Figure 2.6 below) in his article to discuss why technical debt can be caused by prioritization. As per the triangle, every project has three main constraints: scope (e.g., set of features to be implemented), resources (e.g., budget, employees) and schedule (i.e., time). Additionally, the quality of a project in question is represented by the area of the triangle itself. The main message of the visualization is that even though the constraints can be decreased towards the center of the triangle or increased in the other direction, managers should not forget about the implications of changes with respect to quality. Naturally, in order to maintain the quality of the project, if one of the constraints is increased, one or both of the other ones have to be decreased. However, in real life, at least one of the constraints is fixed, while the other ones can be moved freely and contribute to the adjustments made to the quality.

Allman [1] proposed including a fourth corner for technical debt, thus turning the visualization into a square (in an ideal scenario). In his opinion, many managers use this "forever" free fourth corner as the means of applying their own priorities without decreasing the overall quality. In other words, technical debt is the constraint that can "take the blame" for anything done to the other three constraints.

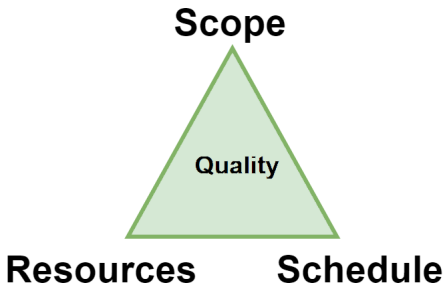

Figure 2.6: The project management triangle

Tom et al. [39] in their article also described an extremely typical form of accruing technical debt: proof of concept projects being transformed into production code. Proof of concept implementations are meant to be thrown away by the very nature of their purpose, but managers perceive it as throwing away already invested time, which is a scarce resource. Proof of concept implementations exist to prove that achieving a certain goal is technologically possible, but nothing more. Therefore, they are usually written in a haphazard way, not considering design best practices or the evolvability and maintainability of the solution whatsoever. Consequently, building a whole system on top of them and deploying it in production is a rather poor decision to make.

Customers

Customers themselves can also serve as a huge source of technical debt. Lim et al. [26] also addressed this issue. Their conclusion was that unfortunately, many customers do not really know what they want or need exactly. As a consequence, if software development organizations do not make a big enough effort to find out every important requirement from their clients, they often end up making assumptions. Similarly, the same thing can be said about markets as well. In the same article, they gave two examples for typical scenarios when customers cause technical debt:

- *Last-minute requirements:* This has to do with the fact that clients usually do not know what they need. However, during the user acceptance testing process — when they actually see a functioning version of the product — they often realize that some of the features need to be changed or new ones added so that the product better fits their requirements. Not surprisingly, if the current state of the implementation is very far from the ideal state in some aspect(s), the difference takes the form of technical debt.

- *Extensive wish list:* Sometimes customers have an extensive list of features, but at the same time, they wish to have the software ready in an extremely short time. At the development team level, this creates massive pressure, which triggers prioritizing actions, thus resulting in technical debt.

Attitudes

Attitudes of individuals — that of managers and engineers equally — have a great influence on technical debt creation. As Tom et al. [39] suggested, there is a general apathy present in the development team, which has to do with the already discussed lack of incentives to avoid technical debt. As earlier mentioned, people might not even work at the company when the debt has to be paid down.

Additionally, managers and technical people have a different attitude and especially risk appetite. This view is also advocated by Lim et al. [26] who stated that managers are generally used to taking risks, therefore, they are less uncomfortable with technical debt as well.

Ignorance and oversight

In their paper, Tom et al. [39] also addressed the issue of ignorance and oversight. While the former comes from the personality of individuals, the latter has more to do with the skills, knowledge, personal life and mental well-being of employees.

Fortunately, the level of ignorance can be reduced by making people understand how crippling technical debt can get. Even though it is not an easy thing to achieve. Similarly, the frequency of oversight-based technical debt creation can be lowered by being methodical, considering ramifications of decisions and taking into account what feature requests might surface at some point in the future.

Processes

The way how recurring everyday tasks at work are handled can also influence the total amount of technical debt that is accrued. In accordance with what Tom et al. [39] indicated, poor communication collaboration practices can introduce some extra debt that would be completely avoidable otherwise. In addition to that, repetitive, but not automated are also prone to technical debt.

Good examples of such debt sources are not hard to find. For instance, not using the right set of tools, such as means of communication tools (e.g., the usage of slow emails instead of instant messaging applications for every

type of communication), visualization tools (e.g., charts) or the lack of code reviews both contribute to increased levels of technical debt.

Software aging

As cited by Brown et al. [7], Parnas [33] described software aging as *"the failure of the product's owners to modify it to meet changing needs"*. This is in line with one of the laws of Lehman & Bélády [23] saying that software solutions need to continuously adapt to the changes of the environment. Failing to do so, makes the software lose some of its utility, and thus the debt of restoring that is created.

On a related note, Ernst et al. [12] added that the drift caused by changing system use is proportional with the age of a system in question. Similarly, bad architectural decisions become more and more emphasized as time passes by and the system ages.

2.4.3 Ranking of technical debt sources

As mentioned before, this subsection provides a more detailed ranking of debt sources. Ernst et al. [12] asked a large number of IT practitioners to choose the top three from a set of sources with respect to their prevalence. The results of their survey can be seen bellow in Figure 2.7. Hatches represent the percentage of time a certain source was picked first, while being picked second is displayed using dashes and the third choices with dots.

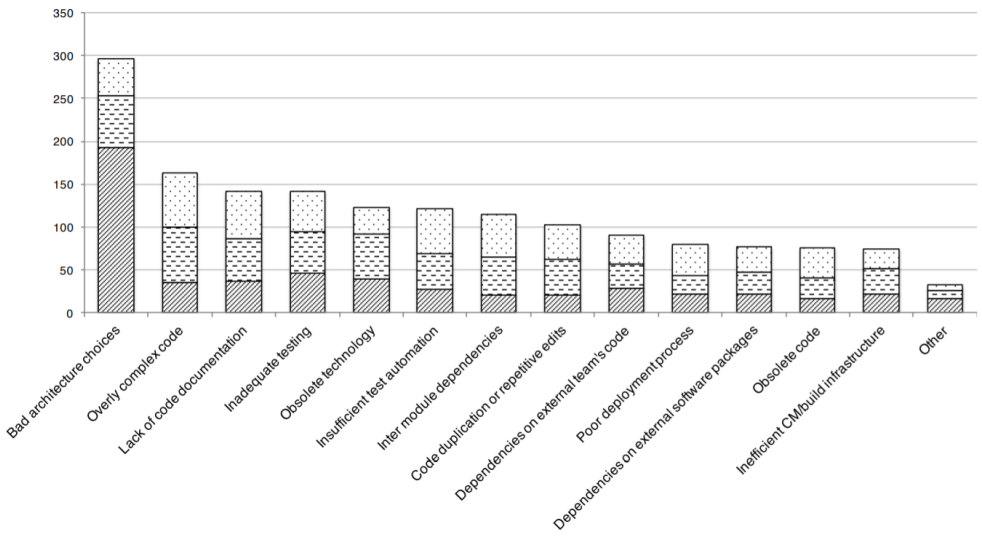

Figure 2.7: Technical debt source ranking by Ernst et al. [12]

2.5 Effects of technical debt

This section of the thesis examines the possible effects of technical debt on a software development organization. As it can be seen, technical debt can influence everyday activities of companies in various different ways. Being aware of these effects can be a useful tool for communicating technical debt related issues toward less technical stakeholders.

2.5.1 Effects on morale

According to Tom et al. [39], many developers think of technical debt management as a "mundane task". Therefore, engineers usually lack the motivation to spend time on it.

As they explained, on the one hand, technical debt has a positive effect on short-term morale most of the time. This is rather understandable, since people in general prefer to make continuous progress and have a tendency for choosing the easier solution for problems. This view was supported by Laribee [22] as well, who said the following:

> *"It's painful when I'm not productive and it's pain that robs me of potential productivity, the so-called "good days at work"."*

Hence, if a developer can save some time and effort by implementing a sub-optimal solution, it certainly has a positive effect on their short-term morale. However, this also depends on the individual and their personal preferences, since some of the engineers can get really frustrated by being forced to do workarounds and other types of poor design.

On the other hand, as Tom et al. [39] pointed it out, one should not forget about the real issue with technical debt. It becomes "painful" only in the long term, when it has to be paid down for some reason. Therefore, its long-term effects on morale are crippling, since instead of experiencing the short-term frustration of doing a task properly, extra annoyance is added by the interest payments. In addition, the extra work has to be carried out working on a rigid and chaotic code base.

2.5.2 Effects on productivity

Tom et al. [39] also examined the short-term and long-term effects of technical debt on productivity. In accordance with their comments, taking on debt temporarily increases the velocity of development, and thus the momentary

productivity of the organization as well, but at the same time, hinders having a good feature delivery rate in the future. This is due to the similarities between high-interest loans and technical debt.

With respect to long-term effects, the existing code base becomes every time harder to modify (i.e., poor evolvability) and — as the participants of the survey of Tom et al. [39] also indicated — development generally slows down. Having brittle software components also goes hand in hand with an increased number of regressions. Self-evidently, diagnosing and fixing these issues consumes a lot of time, thus resulting in a decreased amount of time that can be spent on actual feature development.

Another technical debt-related aspect is inadequate knowledge distribution. When people have to spend an unnecessarily long time understanding and examining the system before beginning implementation, the productivity suffers a decline. Participants of the survey also suggested that if implementing a workaround takes a long time, sometimes realizing the need for one requires even more.

2.5.3 Effects on quality

In consonance with the article of Tom et al. [39], technical debt effects software quality on many different levels. One thing, however, is common for all of them: they can lead to issues even in the short term. Later on, these effects get even worse due to the corresponding interest payments.

They also emphasized that the main quality problems boil down to source code which is hard to read and also understand because of its complexity. Quality issues not only create new defects, but also contribute to existing defects staying hidden from developers.

Tom et al. [39] identified quality issues related to the following categories:

- Extensibility
- Scalability
- Maintainability
- Adaptability
- Performance
- Usability
- Testability
- Supportability
- Reliability
- Security

In addition to the previous categories, Lim et al. [26] also addressed the topic in their publication. They also found that major effects of technical debt involve increased complexity, poor performance, system instability and fragility. Therefore, it is of no surprise that one of the participants of their survey described a typical situation as follows:

> *"[...] you feared that any time you made a change, you were going to cause something else to go wrong."*

2.5.4 Effects on risk

According to Tom et al. [39], technical debt potentially introduces a significant amount of risk to any software development project. Effort estimation becomes extremely hard for even the most experienced IT practitioners because of all the unknown factors and variables that are inherently present due to the presence of technical debt. As a result of that, the development process becomes less deterministic, which is a form of risk.

The researchers also pointed out that the visibility of technical debt in question also has a considerably large influence on the difficulties it can cause. Self-evidently, in case of known technical debt the only complication is caused by the need for estimating the extra work (interest payments). However, in contrast, inadvertently accrued technical debt makes the estimation process even harder, since not even the principal part is visible to employees. This is considered risky, since nobody anticipates the appearance of related issues.

2.6 Technical debt management

This section provides an overview of some of the most relevant technical debt management aspects. Although these areas will be further discussed more in detail in the methodology, mentioning them is beneficial when it comes to understanding the context better.

2.6.1 Overall technical debt management approach

According to Robert C. Martin [37], the basic management approach that should be applied in case of technical debt is essentially the same as the one know from the field of monetary debts, such as taking up a mortgage. More in detail, he talked about having an increased discipline and paying a closer attention to one's spending and accounting. The clean coder also pointed out that the level of discipline should be proportional to the amount of technical debt.

On a related note, Ramakrishnan [35] also emphasized the necessity for increased levels of vigilance. Furthermore, he also identified some other key aspects to technical debt management, such as not using shortcuts, trying not to over-engineer components and refactoring not-quite-right pieces of code whenever it is needed and possible.

Kruchten et al. [21] highlighted the importance of technical debt identification and explicit management. For instance, they proposed the usage of a technical debt backlog, organized in the way that is presented by Figure 2.8.

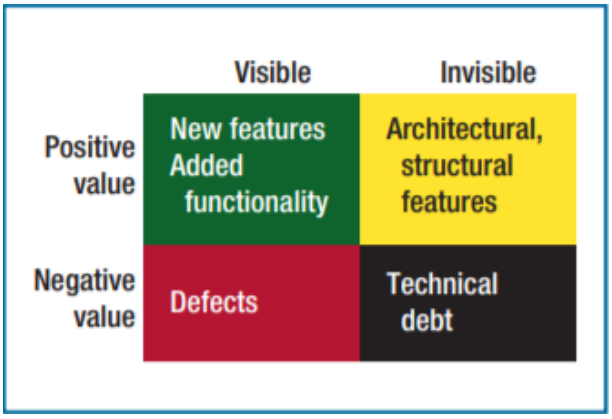

Figure 2.8: Technical debt backlog organization by Kruchten et al. [21]

According to the views of the authors, it is to be considered bad practice that project backlogs in general only contain positive-visible elements (i.e., new features and added functionality) and positive-invisible ones (i.e., architectural, structural features), ignoring defects and technical debt. Although, defects are usually stored somewhere else (e.g., defect database), technical debt is most of the time completely forgotten. In their opinion, this kind of phenomena should not happen and a unified backlog needs to be maintained to facilitate efficient project management.

Laribee [22] put the emphasis on system thinking in his article. In his opinion, the only way of efficiently managing technical debt is to practice long-term, investment-oriented thinking and understanding the difference between projects and products. As he explained, most of the time, software development teams work on a product and not just a project. Even though projects end at some point, successful products live long and maintaining, modifying and extending them is the job of the same organization that developed them in the first place. Therefore, it is of vital importance to think ahead and do development work in the present accordingly. Additionally, every person associated somehow with a given project or product should contribute to the efforts for improvements.

In order to provide some help with cultivating "buy-in" of stakeholders, Laribee [22] also referenced a useful tool proposed by a blog post of Finley [13]. She suggested the usage of a very simple sentence: "evidence DEFEATS doubt", where the verb is used as an acronym for the following good practices:

- **Demonstrate** the real impact on development.
- Give **Examples** of cases when technical debt had a negative effect on feature development.
- Prove with **Facts** that software development is suffering from technical debt, such as being unable to meet deadlines.
- Avoid the usage of jargon using **Analogies** to avoid losing the attention and understanding of less technical people.
- Use **Testimonials** — or anecdotes lacking the budget to hire someone — to show how other companies benefited from implementing a proper technical debt management strategy.
- Support every statement with up-to-date **Statistics**.

Bohnet & Döllner [4] — in consonance with Kruchten et al. [20] — described the management of technical debt as finding the balance between internal and external qualities. They defined internal qualities as aspects such as *"conformance to architectural/design principles, modularity and clearly defined interfaces, and code complexity"*, in other words all the things that are only visible to developers. In contrast, external qualities can be perceived by others as well, such as *"post-delivery defects detected by customers or the number of implemented features per iteration"*.

It is also worth mentioning that the process of finding this balance is often very difficult, which makes technical debt management rather complicated as well. As a rule of thumb, external qualities are the ones used to measure the overall quality of a product, which introduces a significant bias. In addition, what further complicates the situation is that return on investment is immediate for these qualities, while that of internal ones take more time. This is the reason why long-term quality often gets ignored for the sake of short term gain, leading to increased pressure in the future.

In terms of difficulties, Power [34] identified a set of other obstacles that one can face when trying to manage technical debt. These are summarized by the following table:

Challenge	**Description**
Definition	Since numerous different interpretations of the metaphor exist, it is important to make sure that everybody has the same understanding about the term.

Quantification	It is not always straightforward and easy to find the best way of quantifying and measuring technical debt. Some of the examples given by Power [34] were to measure team capacity or feature development velocity.
Visualization	Companies should use efficient visualization techniques to facilitate the communication of debt.
Tracking	Once debt is detected, it also needs to be tracked somehow so that it stays under control. Doing so also takes some extra effort.
Neglecting debt	According to Power [34] — and in accordance with what was discussed earlier —, even though neglecting technical debt for one release has no noticeable impact on productivity, as newer releases come, the effect keeps becoming more and more significant.
Root cause ignored	When bugs are detected, often just the most immediate causes are fixed, which results in experiencing returning defects. Therefore, it is important to always find the root cause, because it can easily be technical debt.
Costs of delay	Many companies have a hard time understanding the real costs associated with technical debt. Possible rework can equal to an extremely high number of extra work hours.

Finally, another technical debt management-related aspect concerns the set of tools available. As Ernst et al. [12] pointed out, there was still a need for good practices and tools to handle technical debt and the situation has not changed much ever since they published their article.

2.6.2 Technical debt-related activities

In order to create a methodology, it is beneficial to identify and define some technical debt-related activities. Li et al. [25] introduced the following list of activities:

- **Identification:** Find existing technical debt of companies using a set of tools, such as static code analysis or checklists.

- **Measurement:** Introduce techniques for quantifying the amount of already existent or newly introduced technical debt.
- **Prioritization:** One of the most important activities of technical debt management, since most of the time, there is a need for an ordered list of debt at some point during the decision making process.
- **Prevention:** Its main purpose is to avoid incurring further debt on top of the already accumulated amount, thus, facilitating the efficient reduction of overall technical debt.
- **Monitoring:** Once technical debt is identified, it is of crucial importance to keep unpaid debt observed in order to avoid letting it get out of control. This activity aims to monitor changes of existing debt.
- **Repayment:** Refers to the act of eliminating technical debt instances. This can be done using various techniques, such as refactoring or re-engineering.
- **Representation/documentation:** Probably one of the most important activities which usually does not get enough attention. It focuses on making technical debt well-documented and visualized, so that every stakeholder can understand given situations and the implications.
- **Communication:** This activity goes hand in hand with the previous one, as it also serves the purpose of letting stakeholders know about difficulties related to technical debt so that the necessary measures can be taken.

As it can be seen, all of the activities mentioned above are essential. Therefore, they will be further discussed as part of the proposed methodology.

Chapter 3

Methodology

This chapter forms the core of this thesis. It introduces a methodology for technical debt reduction and management in eight steps. The methodology is designed to be practical and easily applicable in an SME environment.

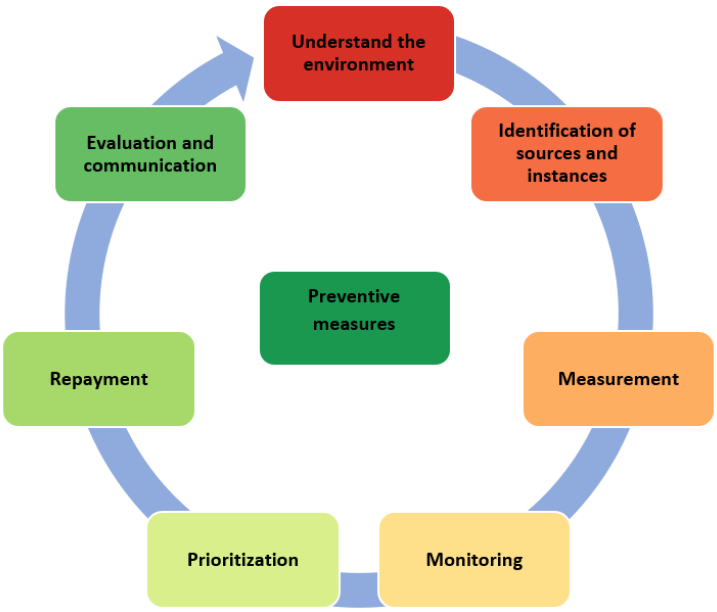

Figure 3.1: Overview of the methodology

As it can be seen from Figure 3.1 above, the methodology is intended to be used in an iterative way. In principle, it consists of 7 plus 1 activities. While preventive measures need to be taken in a continuous way, the rest of the steps

are to be executed sequentially, since they build upon each other. However, since software projects continuously evolve and change, the technical debt management strategy of SMEs also need to take these changes into account and revisit each activity periodically, every once in a while.

Unlike in already existing methodologies, in this methodology, understanding the environment, the communication of technical debt and prevention measures also have an important role. In the upcoming sections of the document, several alternatives are presented for each of the actions depicted by Figure 3.1.

3.1 Understand the environment

Any person that is given the task to reduce the technical debt of an SME, needs to start the process by examining the overall profile of the company first. Even if the person in question is not a new employee, in order to succeed, he or she has to understand how different aspects of a company work. For instance, many developers have little knowledge about communication practices between managers and clients, since it is not something essential for their jobs. In a similar fashion, managers and clients tend to be less aware of technical details, saying that they prefer to leave them for technical people. However, — as Norberg [31] also pointed it out — a general understanding of business activities and the technologies involved is very important to the success of software projects. Since a large number of technical deficiencies tend to be of system-wide nature (affecting organizations as a whole), technical debt management can also benefit from this kind of knowledge.

Consequently, as one of the first steps, it is worth to understand the profile of the company more in detail. As part of the process, one should examine:

- What is the main offering of a company (e.g., a product, software as a service)?
- What are the main business activities of the SME in question?
- Which projects have the highest revenue? Knowing this piece of information can be especially useful when prioritizing technical debt items.
- What are some limitations and dependencies (i.e., boundary conditions)?

The next relevant aspect that can influence a company's relationship with technical debt is whether they work on innovation projects or not. Not

surprisingly, if they do, the company is inherently more prone to accruing technical debt than others are. This is due to the fact that it is particularly hard to see all the requirements at the beginning of a project and the overall goals can also change in time.

Furthermore, reviewing the main stakeholders of projects is also a good idea. Discovering the roles and hierarchy of stakeholders can help to establish efficient communication practices for instance. In other words, it is of essential importance to discover who are the people that need to understand the dangers of technical debt in the first place, in order to facilitate the effective management and reduction of it.

On a related note, knowing what their backgrounds are also helps to communicate with them in the most fitting manner. While some of them might have a managerial background, others might be more focused on technical aspects of projects. Therefore, it is also beneficial to learn about the skills and responsibilities of each of them. As earlier discussed in section 2.4, this difference can result in very different risk avoidance attitudes, which complicates the management of technical debt even more.

Just like it is necessary to understand the network of people, it is also required to understand the technical structure of projects. In other words, the purpose of each component should be identified, alongside with their relationships towards each other. Furthermore, discovering the level of abstraction and modularity should also form part of the process, since these aspects can easily serve as the source of technical debt.

Finally, collecting a list of the technologies involved can also contribute to the success of identifying technical debt sources (see section 3.2). In order to understand what they are used for, the documentation — if it exists — can contain useful insights. Naturally, documentation can be useful throughout the entire first step of the methodology. Therefore, this supports the idea that enforcing proper documentation practices should not be taken lightly, since poor documentation can result in scenarios where new employees are forced to "re-discover the wheel" over and over again, while trying to understand the system.

3.2 Identify technical debt sources and instances

A technical debt reduction methodology cannot exist without a step that is dedicated to the identification of it. Since many sources of technical debt exist (as explained in section 2.4), its identification is not necessarily a very

straightforward process, even though there exist some tools and approaches that can help with it.

Identification starts by raising awareness about technical debt, since many software practitioners do not take related issues seriously enough. In most of the cases, this is due to a lack of general knowledge and understanding. Additionally, there tend to be competing opinions about the definition of technical debt among people. Thus, as a first step, employees and stakeholders need to be educated about the phenomena, also reaching a consensus about the definition of technical debt within a given SME.

Once everybody has the same understanding about technical debt, the next thing to do is determine how related issues are identified. According to the findings of Ernst [12] presented below in Figure 3.2, the most prevalent way of identifying technical debt is either by doing so during retrospectives or by implicitly registering instances in the backlog. Conversely, a rather low number of technical debt tools are used for identification. This can be explained by the fact that tools only exist for identifying source code-related debt items.

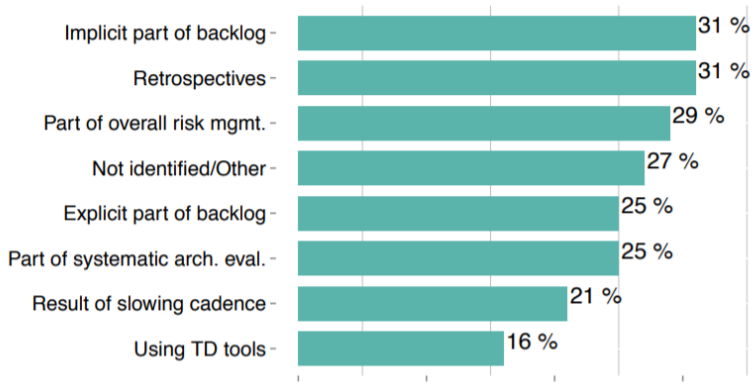

Figure 3.2: Ways of identifying technical debt by Ernst [12]

As it can be seen from the chart, a significant percentage of IT practitioners follow the *"fail first, identify technical debt as a cause"* approach, since the "Not identified/Other" category is rather substantial as well. Naturally, this is not the recommended way of treating the question of technical debt. Without doubt, the extra effort needed to eliminate the accumulated technical debt can become crippling and completely paralyze feature delivery as time passes by. It is good to bear in mind that interest payments do not only manifest as defects, but also take the form of other difficulties as well. Therefore, it is beneficial to identify and track technical debt as soon as pos-

sible, so that necessary measures can be taken in case it becomes a threat to the organization.

Social cues can also indicate elevated technical debt levels of a company. For instance, if developers try to avoid working on certain parts of the code base or people leave the company due to low morale, one can be sure that the root cause is technical debt. Therefore, it needs to be identified and properly managed. Furthermore, some other typical indicators of technical debt is having god objects, dead parts of code, spaghetti code and frequent test failures due to brittleness of the system.

On a related note, deliberate technical debt should be always self-admitted. As already pointed out in section 2.1.1, technical debt is neither inherently bad or good, it is sometimes inevitable. However, keeping it hidden or neglected is to be considered bad practice. Therefore, developers should be encouraged to be transparent about the technical debt that they potentially take on deliberately, so that it remains under control.

However, not only deliberate debt exists and being aware of inadvertent debt is also key in technical debt reduction. For instance, Tufano et al. [40] suggested that the majority of code smells is introduced by the very first commit or as part of refactoring commits. This implies that a huge amount of debt could be avoided just by double-checking the quality of commits.

As the most basic approach, technical debt can be identified using simple checklists. However, these lists can only provide some general guidelines and a set of clues for doing so. Therefore, some additional approaches are discussed in the following subsections.

3.2.1 Code analysis

Li [25] proposed carrying out code analysis as means of identifying technical debt. Its main goal is to find source-code related weaknesses of a software product that can potentially turn into vulnerabilities or make the code base difficult to understand, manage and extend. Furthermore, it is also important to emphasize that the process is intended to be completely automated and it can possibly return code metrics as well. Code analysis can be equally done in a static and a dynamic way as well.

3.2.1.1 Static Code Analysis

Static Code Analysis (SCA) focuses on verifying whether the source code is compliant with a certain — predefined or customized — set of coding rules or not. This type of analysis is carried out during the implementation phase of the software life cycle.

Nowadays, SCA also forms part of modern IDEs, as immediate feedback can help developers notice their mistakes at the moment of making them. A good example of this use-case would be the usage of code linting tools. Additionally, when run separately, analysis tools can yield different kinds of code metrics as well, which provide further insights into the quality of the source code.

The publication of Marinescu [27], can help in understanding how SCA tools can identify technical debt. For instance, the combined use of the most basic object-oriented metrics — coupling, cohesion, complexity and encapsulation — can indicate the presence of a set of object-oriented programming-related issues, such as god classes, data classes or methods with intensive coupling.

3.2.1.2 Dynamic Code Analysis

Dynamic Code Analysis (DCA) is carried out during execution time of a program. There are certain aspects (e.g., memory error detection) that can be only examined using this dynamic approach. DCA is usually used during the testing phase of the software life cycle.

With respect to tools, it is very important to instrument the source code in such manner that does not introduce noise or bias in the results of the analysis. Additionally, it is good to remember that the code coverage of DCA depends on user interactions, which can easily result in parts of the code being left unexamined.

3.2.2 Dependency analysis

Li [25] also suggested the usage of dependency analysis, as it has a vital role in escaping the so-called "dependency hell". Not managing dependencies appropriately can certainly cause serious headaches to the development team, since resolving dependency-related issues is not always evident. Therefore, it is not by mistake that these issues were given the aforementioned name. Some of the most frequently encountered difficulties are:

- **Unnecessarily large number of dependencies:** There is no universally recommended limit, as it greatly depends on the characteristics of the platform in question. However, as a rule of thumb, it is safe to say that the number of dependencies should be kept at a minimum level. For instance, iOS developers should ideally restrict themselves to using maximum six libraries. Furthermore, when the need for the usage of a library arises, the order of preference should be: native libraries

first, own implementation second and third-party implementations last. This is owing to the fact that third-party packages come with inherent risks in terms of security, licensing and performance.

- **Conflicting versions:** When two different versions of the same library are required at the same time, developers might encounter a dependency conflict at the moment of introducing the second one.
- **Transitive dependencies:** A transitive dependency refers to dependencies that exist due to the mathematical transitiveness property. In other words, if component A directly depends on B and B on C, A also depends on C implicitly, which is naturally not an issue per se. However, since explicitly defined dependencies possibly depend on other libraries as well, it is possible that when a given developer starts using dependency D (which also depends on C), he/she might forget to define C as an explicit dependency. If later on, dependency A is no longer needed and it is removed (alongside with B and C), dependency D stops working all of a sudden, which makes the project less deterministic.

3.2.3 Analyzing statistical data

Oftentimes statistical or metadata of projects can be a good indicator of the existence of technical debt. Self-evidently, most of these approaches are unable to name the exact source of technical inefficiencies, but detecting potential issues is always better than thinking that everything is the way as it should be.

For instance, a burn down chart — well-known from Agile software development — can be one of these indicators. If the number of unfinished tasks is high or has an increasing tendency at the end of iterations (e.g., sprints), the project suffers from productivity loss, which can easily happen due to technical debt. Among others, Li et al. [25] also introduced a similar indicator called ANMCC (average number of modified components per commit), which could be utilized to detect architectural technical debt. If this number is high, it means that the *separation of concerns* principle was not applied properly. On a related note, this concept could be extended to tasks instead of commits as well.

3.2.4 Identification by experts

Unfortunately, not every type of technical debt can be automatically identified. As already mentioned before, most of the tools can only handle the identification of source code-related issues.

Therefore, in many cases, identification has to be done manually by experts, such as software architects or developers. Most of the time, they can only rely on their experience and software engineering knowledge. Typically, this kind of debt identification does not happen as a separate activity, but as a by-product of working on the code base instead.

3.3 Measure technical debt

Devising ways of measuring technical debt is probably one of the most difficult tasks to do. As detailed in the literature review part of the document (chapter 2), a significant number of debt types exists which makes their measurement rather complicated, since every type requires potentially different ways of quantifying them.

Debt can be measured both in an automated and a manual fashion:

- **automatically:** using different types of models, computed metrics and ordinal scales of measurement
- **manually:** relying on estimates given by IT practitioners

Measurement efforts aim to convert technical debt into actionable information. Based on different expectations, this can result in having a number or severity assigned to technical debt. While the most commonly used numeric measurement types are monetary costs and the number of man-hours that are required to eliminate technical debt, severity is usually measured using ordinal scales (e.g., low, medium, high). Although both approaches can result in somewhat inaccurate measurements, errors can be mitigated with the help of historical data.

Depending on the intended usage of measurement data, it is important to evaluate which types of technical debt should be measured and which can be simply excluded from the scope of measurements. In addition, it is also noteworthy that the frequency of measurement might vary for different types. For instance, the high-level architecture of a system tends to change slower than other characteristics, and thus architectural debt can be measured with a lower frequency.

There are three main measurement approaches recurring in the corresponding literature: the usage of technical debt measurement models, calculating metrics and leaning on the insights of software practitioners. Therefore, these three are described below.

3.3.1 Models

Models provide a systematic way of measuring technical debt. Therefore, this section introduces the most well-known solutions. Technical debt measurement models can be categorized based on measuring only principal, only interests or both. While the interest-based approach places the emphasis on the costs of recurring extra work (typically identified and measured during retrospectives), the principal-based approach advocates the elimination of both one-time and recurring work. Hence, it is of no surprise that the models introduced in the following sections also address the task in the latter way.

3.3.1.1 SQALE model

The name of this model stands for *"Software Quality Assessment based on Lifecycle Expectations"*, therefore, it is used by many source code analysis tools. The SQALE method utilizes both a quality model and an analysis model. Its quality model consists of three hierarchical levels as shown in the Figure 3.3 borrowed from Letouzey [24]. These levels are: characteristics, sub-characteristics and source code-related requirements. These requirements need to be customized for software context and programming language.

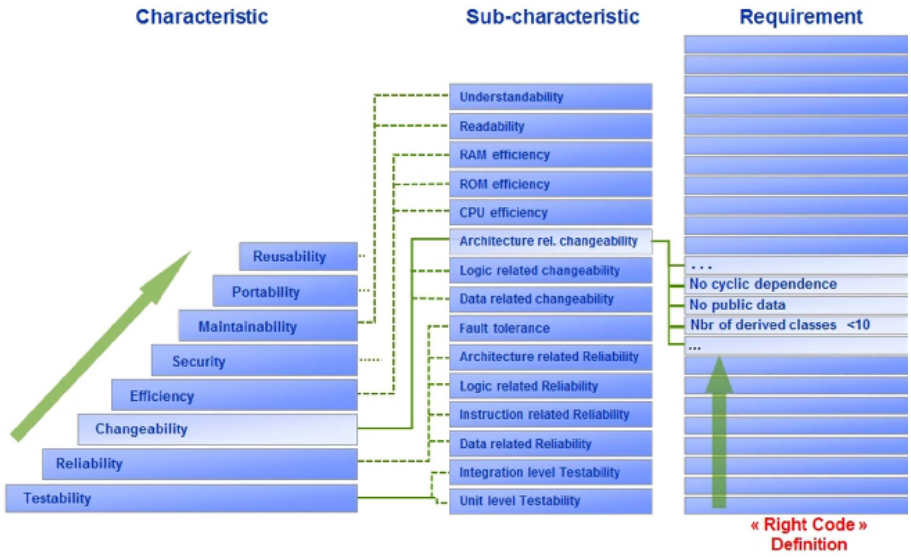

Figure 3.3: Visualization of the SQALE quality model by Letouzey [24]

The SQALE method also ships with an analysis model that can be used to calculate remediation indices for different artifacts, such as a module, a

file or a class. Since the method aims to measure technical deficiencies as the difference between the current state and the quality target of the source code, remediation indices have a crucial role in the analysis process.

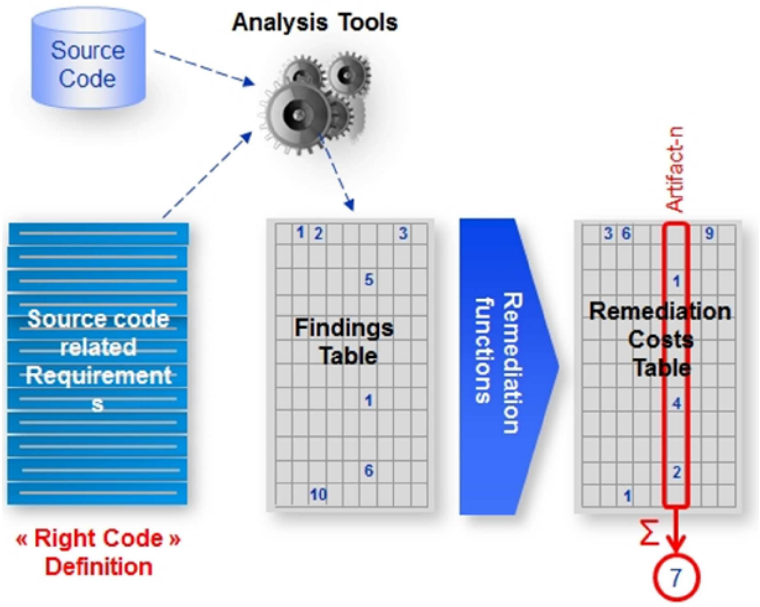

Figure 3.4: Visualization of the SQALE analysis by Letouzey [24]

Letouzey [24] also created a visualization of this analysis model (Figure 3.4). Based on that, the process of calculating the aforementioned remediation indices can be broken down into the following set of steps:

1. Validating the source code against the requirements that were earlier defined in the third level of the quality model. Every violation is counted in a matrix, where each column corresponds to an artifact and each row represents a source code quality requirement.

2. A so-called remediation function is applied to the matrix, which makes sure that violations of different requirements are properly normalized. This function needs to be defined in a customized way by the organization and it makes sure that non-conformities of different types can be measured and compared with each other.

3. As a final step, remediation indices are aggregated either for every artifact separately (e.g., separate files) or for the tree structure of the quality model (e.g., SQALE Testability Index, STI). These indices rep-

resent remediation costs and a global SQALE quality index (SQI) is also calculated.

3.3.1.2 CAST model

This model was developed by a leading software intelligence company called CAST and it is used in their Application Inteligence Platform (AIP). Before the model can be applied, the source code needs to be parsed and validated against a rule set. According to Curtis et al. [10], the model evaluates over 1200 rules with the goal of detecting violations of good architectural and coding practice. This model aims to measure the monetary costs of the existing debt directly.

As explained by Curtis et al. [10], CAST focuses on the principal part of debt. According to the model, this part of technical debt depends on the number of "must-fix" violations, the time needed to fix them and the costs of doing so. Since not every issue has to be fixed, the CAST model uses the following formula to calculate technical debt:

Principal =

(number of ***high-severity*** *violations)∗*
(percentage to be fixed) ∗ (average hours needed to fix) ∗ ($ per hour)+
(number of ***medium-severity*** *violations)∗*
(percentage to be fixed) ∗ (average hours needed to fix) ∗ ($ per hour)+
(number of ***low-severity*** *violations)∗*
(percentage to be fixed) ∗ (average hours needed to fix) ∗ ($ per hour)

As it can be seen, violations are grouped into three severity groups. As per the publication of Curtis et al. [10], "must-fix" percentages can be initially set to 50%, 25% and 10% in decreasing severity order. However, this and the hourly cost parameters need to be tailor-made for different projects and organizations. Hourly costs in particular depend on geographical location and the experience level of the person carrying out given rework. With respect to the time component, it is good to keep in mind that it should involve the time spent on the analysis of the problem, understanding the source code, finding the solution, considering potential side-effects, implementing the solution, running tests and releasing the fix.

3.3.1.3 Counting the number of violations

The most simple and most straightforward model that can be used is the one that simply counts violations. Even though it is a very simple approach, this section provides some recommendations to take into consideration. It is also worth mentioning that when it comes to communicating technical debt towards non-technical people, it is probably better to use an approach that results in more comparable measures, such as the actual monetary costs of eliminating the existing debt.

Violations can be efficiently counted for instance by defining thresholds for certain metrics and grouping thus counted instances according to their types. However, one should not forget that this model can only be used for obtaining rough estimates of the accumulated technical debt levels. Additionally, as it has been already pointed out, since the resultant numbers are not normalized or weighed in any way, they cannot be used as the means of comparing technical debt items to one another in order to prioritize them for instance.

3.3.2 Metrics

The usage of metrics is another efficient way of measuring technical debt. Utilizing the right selection of tools, they can be calculated automatically and periodically, thus monitoring the overall health of software projects.

The following list below aims to enumerate and describe the most important metrics that can be used for technical debt measurement. Since it is a common approach to assign a severity to each of them, initial recommendations are also included with respect to defining the status thresholds (Normal (N), Warning (W) and Critical(C)). Unfortunately, it is impossible to provide a universal configuration for severity thresholds, since they need to be adjusted to the needs and technical debt management strategy of every organization. The aforementioned metrics are explained below.

3.3.2.1 Code duplication

It measures what percentage of the total number of lines are “copy & pasted”. Code duplication is known to cause technical debt, since the same lines of code are present in more than one location, which means two things:

1. If the lines in question introduce a bug, the same defect is present in multiple parts of the code base. Once one of them is discovered, often the other one(s) are left unattended, since it might be a completely different developer that fixes the code, which can lead to fixing the

same mistake over and over again, just in different parts of the code base.

2. If duplicated lines need to be changed as part of feature development work, the same issues as before might arise, due to the need of applying the changes to every instance of the duplicated logic.

In an ideal situation, these lines of code should be placed in a re-usable method the very moment they are used for the second time. However, in real life, a "rule of three" can be observed: the first time the code is implemented without introducing technical debt; then the code is consciously duplicated and finally, when the need arises to duplicate code for the third time, the common parts are extracted into methods. Unfortunately, this last step requires some extra work (interest payments) in the form of refactoring and it should be avoided by all means.

Threshold recommendations:

- Normal: less than 5%
- Warning: between 5% and 10%
- Critical: more than 10%

3.3.2.2 Overall coding best practice rules

This metric measures what percentage of the coding rules are being respected. Due to the fact that these rules are based on best practices, the set of rules greatly depends on the programming language used and the preferences of the development team. Although, some more generic rules can be defined as well, for instance to ensure the compliance with object-oriented best practices (e.g., by measuring the number of classes with low cohesion).

For instance, some teams might prefer having very strict naming conventions, while others do not place a huge emphasis on them. As a direct consequence, their ideal rule sets are also different. Fortunately, most of the available tools come with predefined rule sets, however, they also provide the freedom of disabling them separately and the possibility to create custom rules.

Threshold recommendations:

- Normal: more than 80%
- Warning: between 60% and 80%
- Critical: less than 60%

3.3.2.3 General documentation

This metric measures to what extent the source code is documented, usually in the form of source code comments. To many software practitioners this might sound as one of the most debatable metrics, since it is hard to be measured in a meaningful way, it can not measure the quality of documentation objectively and developers claim to produce self-documented code anyway.

Writing comments can be a complicated task to do, since it is hard to estimate what is worthy of mentioning. While too few lines of comments can make the source code hard to understand for others, too many comments can also have the same effect. As a rule of thumb, trivial steps should never be mentioned, but non-trivial algorithms should always be explained in comments.

Threshold recommendations:

- Normal: more than 20%
- Warning: between 15% and 20%
- Critical: less than 15%

3.3.2.4 Interface documentation

This metric measures what percentage of interfaces are properly commented. It is particularly important to document them, since they are elements that are used by many other components in order to implement a given functionality. Naturally, when implementing a new feature, it is expected to easily understand which interface can be used for a given task and what sort of information needs to be passed to its methods (also defining the characteristics of the arguments). Making these pieces of information explicitly available to everyone can facilitate the work of others in a significant way.

Threshold recommendations:

- Normal: more than 90%
- Warning: between 75% and 90%
- Critical: less than 75%

3.3.2.5 Method complexity

The most typically used metric for this purpose is the Cyclomatic Complexity (CC) developed by Thomas J. McCabe, Sr. [29]. This metric measures the number of linearly independent execution paths in the source code of a program.

This metric is defined to show what percentage of methods have a CC larger than 10. Methods with a high cyclomatic complexity require a larger effort from people that are trying to work with them, therefore keeping the metric at a low level can facilitate more efficient work in general.

There is a well-known notion related to complexity: the human brain is better at using and understanding a complex system of simple things than a simple system of complex components. Hence, this notion validates the usage of this metric.

Threshold recommendations:

- Normal: less than 5%
- Warning: between 5% and 10%
- Critical: more than 10%

3.3.2.6 Test coverage

Test coverage is one of the most important metrics nowadays. It describes the degree to which the source code is exercised when running a certain set of tests (e.g., unit tests, integration tests or regression tests).

More and more people realize the importance of software testing and adapt approaches such as Test-Driven Development (TDD) as they can help to detect deficiencies of our program codes at an early stage, thus minimizing their negative effects. However, a significant number of projects still suffer from painfully low levels of test coverage due to various project constraints and human carelessness, such as time pressure or lack of sufficient budget.

Threshold recommendations:

- Normal: more than 80%
- Warning: between 60% and 80%
- Critical: less than 60%

3.3.3 Measurement by experts

As it has been introduced in case of technical debt identification, measuring the amount of the accumulated technical debt can also be done relying on estimates of experts. It is also worth emphasizing that in order to give accurate-enough estimates, these experts need to have a deep understanding of both the technical and managerial aspects of projects.

For instance, even though one can introduce metrics for comment density, measuring documentation debt can be most efficiently done by actual people, since comment density is not able to point out the shortcomings of architectural documentation to start with. Additionally, experts can also take into account the salaries of developers when creating an estimation of monetary costs of technical debt.

3.4 Monitor technical debt

This step cannot be missing from any good technical debt reduction strategy, since organizations should not lose track of the already identified technical debt that they accrued. It would be just as irresponsible as taking up loans in our everyday lives and just forgetting that they ever existed. The more in detail debt can be monitored, the easier it becomes to make decisions about its management.

3.4.1 Monitored information

Some general guidelines with respect to selecting the information that is worth monitoring are detailed below:

- Technical debt items should be **identifiable** and it can also be beneficial to discover **relationship between certain items**. This makes communication easier.
- To further facilitate communication, the exact **conditions and characteristics of technical debt should be described** in a concise, yet clear way.
- Since the **explicit type of technical debt** has implications for its severity (and therefore, for its priority as well), the technical debt type should also be tracked.

- Storing the **actual and up-to-date status** of technical debt items in some form can be a good idea to better see the overall health of the project.
- Knowing the **date of identifying a given debt item** can also help to evaluate overall trends (e.g., in order to showcase the benefits of managing technical debt).
- The **impact of leaving an item unattended** also contributes to the success of prioritizing debt.

3.4.2 Implementation of a monitoring process

Martini et al. [28] addressed the topic of tracking technical debt in their article. As part of their thorough study, they also identified some key elements to implementing a successful and maintainable tracking process.

First of all, they indicated the necessity of having a "champion" of technical debt monitoring. This person should fulfill the role of raising awareness and advocating the adoption of monitoring practices. The "champion" can be of multiple roles: an experienced developer, a software architect or a manager, just to mention a few.

Secondly, workshops should be held, so that everybody understands the concept and goals. Forcing people to register technical debt instances without them understanding the benefits is not a viable option and it normally leads to haphazardly registered details.

Thirdly, as another resource, some time should be set aside to begin monitoring projects. For example, tools need to be configured and the exact tracking information of interest has to be specified. Although, the time spent on these steps cause a loss of productivity in the short term, they are beneficial in the long term.

Fourthly, they also highlighted the importance of guaranteeing the availability of the required budget. In order to do that, it is of crucial importance to involve management people of the organization, making them understand the importance of the cause.

Lastly, the benefits of tracking debt need to be shown to the management. As it was mentioned before, it requires money and extra effort, hence, it needs to be shown that resources are not just wasted for nothing.

Li et al. [25] also addressed the topic of technical debt monitoring in their publication. They identified the below listed five approaches:

- As already mentioned before, **alerts can be configured** for the event of measured metrics **reaching a certain threshold**. When such an

alarm is triggered, the necessary technical debt reduction steps can be taken.

- Dependencies can be used to **trace the propagation** of the negative effects of technical debt.
- It is also a possibility to have **planned checks** and periodically carry out measurements.
- Another approach relies on the fact that **technical debt affects quality attributes** of software projects. Therefore, technical debt can be also monitored by **periodically evaluating attributes**, such as stability, reliability or flexibility.
- **Identifying trends** can also be a form of monitoring technical debt. However, this naturally requires having periodical measurements with a high-enough frequency, so that it makes sense to plot the data.

3.4.3 Types of monitoring tools

Martini et al. [28] also discussed what tools the participants of their survey used for technical debt monitoring. They obtained the following results, in order of prevalence:

1. **Backlogs:** As the most prevalent solution, participants reported the usage of backlogs. Technical debt can be tracked mixed with the items of the feature backlog or it can also have a separate backlog, exclusively for this purpose.
2. **Documentation:** The second most popular way according to their studies was maintaining text documents, spreadsheets or wiki pages.
3. **Static analysis tools:** These tools can automatically identify, measure and track technical debt. Although, there are commercial or open-source tools available (e.g., SonarQube or the CAST AIP), as reported by Martini et al. [28], some companies opt for implementing their own tools and customizing them for the metrics that they really need.
4. **Issue tracking system:** Creating tickets for technical debt items, just like it is done for bugs is also an option. Normally, in these cases, participants mentioned that they assigned a low priority to these tickets, thus keeping them somewhat separated from feature-related issues.

5. **Comments:** This option is clearly less effective than any other in the list. As stated by Martini et al. [28], these comments usually take the form of "TODO" comments, which are convenient for developers. However, they cannot possibly form the basis of a good technical debt management strategy.

3.5 Prioritize technical debt and make decisions

Once the organization identified technical debt and also created means of measuring and monitoring debt items, it is time to make decisions about them. Due to the existence of the already discussed project constraints (scope, resources, schedule) it is impossible to pay down every single bit of technical debt.

However, in accordance with what was explained in the literature review part of the document (chapter 2) — unlike in case of monetary debt — repayment is not even necessary. Therefore, technical debt instances need to be prioritized. This section introduces key factors to consider before making a decision and it also presents a list of prioritization approaches.

3.5.1 Key factors to consider

When creating the action plan of addressing technical debt, a whole series of technical and business aspects need to be taken into consideration. To start with, it needs to be determined to what extent the prioritization can be done objectively. Sometimes — due to the lack of metrics and other measurements — prioritization needs to be done by experts as well. In such cases, besides the earlier mentioned experience and technical know-how of the experts, they can only rely on a hunch in those cases. Therefore, it is recommended to assure that prioritization can be carried out based on strategically collected data. Otherwise, when it is done by experts, employees from both the managerial side and the technical side need to have their say in the decisions that are made.

Determining what factors should form the basis of prioritization is somewhat context-dependent. However, Ribeiro et al. [36] in their publication shared some possible prioritization criteria. All of these were collected by mapping several studies concerning decision criteria for technical debt repayment. As part of this document and partially based on their results, the following prioritization-related criteria are recommended to be considered:

- **Monetary costs** (*Which technical debt instances demand the highest and most frequent interest payments?*)
- **Technical impact** (*Are there technical debt items that on their own or together with others paralyze feature development?*)
- **Ease of repayment** (*Which debt instances can be eliminated the most easily?*)
- **Opportunity costs** *(What kind of improvement opportunities are blocked unless technical debt is payed down?*)
- **Expected usage** (*When will be the refactored part used according to current expectations?*)
- **Impact on customers** (*What is the level of impact on the customer? Do they experience any direct effects?*)
- **System age** (*How old is the system? Will the project be kept alive for a long time still?*)
- **Type of debt** (*What is the source type of the debt?*)
- **Deliberateness** (*Were technical debt instance purposefully created or do they exist inadvertently?*)

3.5.2 Prioritization approaches

Selecting a prioritization approach is not an easy thing to do, even though many authors and software architects described their own strategies before. However, many of them introduced complex processes, some of which require a deep understanding of financial concepts as well. Therefore, to better match the scope of the thesis and to keep the ease of use of this methodology, only some of the simpler approaches are showcased.

3.5.2.1 Cost-benefit analysis

Both Li et al. [25] and Seaman et al. [38] mentioned this approach, since it is one of the most simple ones. As its name suggests, it tries to prioritize technical debt items by examining the relationship between the costs and benefits of repaying a given item.

As Seaman et al. [38] explained, in the first iteration of prioritization, it is enough to use coarse estimations for both variables on a scale of 1 to 9 for instance, which can be later on followed by finer-grade, real-world estimates

(e.g., monetary costs and revenues). As it can be seen from the example given by the authors (see Figure 3.5), technical debt items can be organized into a simple coordinate system where the two axes correspond to costs and benefits.

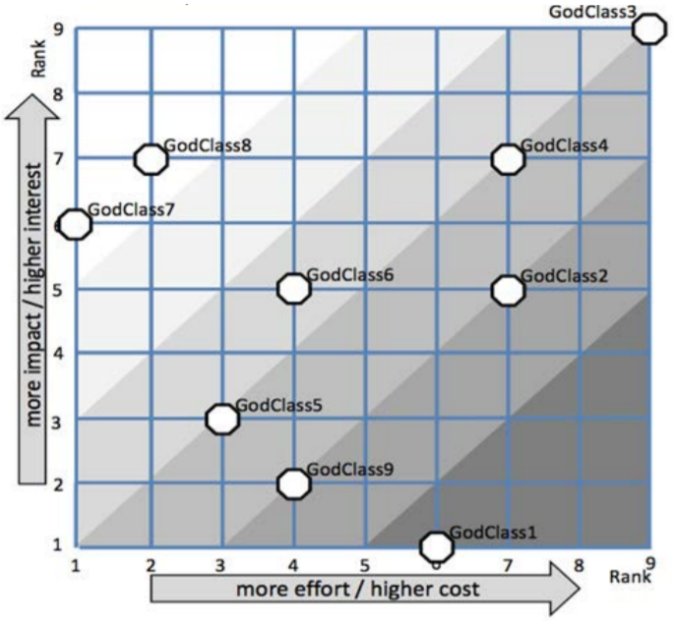

Figure 3.5: Cost-benefit analysis example by Seaman et al. [38]

While the costs can be measured as the extra work effort, benefits can be thought of as eliminating the impact of technical debt items (the higher the impact is, the more beneficial the elimination of an item is). Therefore, the prioritized order of technical debt items can be read from the coordinate system by reading them in order from the upper-left corner towards the lower-right corner. In other words, starting with those technical debt instances that have a high impact and are easily eliminated (GodClass7 and GodClass8) and finishing with those that have a low impact and require a large effort (GodClass1).

3.5.2.2 High remediation costs first

Li et al. [25] also mentioned an approach, which prioritizes technical debt items with respect to their remediation costs. Remediation costs involve every cost that is related to the extra work needed to eliminate a given technical debt instance.

High-remediation-cost technical debt items can be often categorized as architectural technical debt. If teams start with items that have low remediation costs, they risk doing low-level refactoring work on parts of the source code, which might have to be completely changed anyway, when the high-remediation-cost items are reached. Thus, the risk of wasting time on fixing minor issues first is considerably high.

3.5.2.3 High interests first

Another simple and easily understandable way of prioritization was also mentioned by Li et al. [25]. Unlike the other two approaches cited in this master's thesis, instead of examining the principal costs, it tries to give higher priority to those items that demand the highest amount of interest payments.

It is a well-known fact that interest payments represent recurring costs. Hence, the underlying principle of this approach is as follows: the sooner the re-occurrence of payments is stopped, the less effort and money needs to be spent on them.

3.6 Repayment

After having presented some prioritization considerations and approaches in the previous chapter, as the next step, it is time to address the topic of technical debt repayment. First of all, this chapter discusses the importance of continuous repayment. Secondly, it also provides a description of the various ways of retiring technical debt.

3.6.1 Continuous repayment

Just like monetary debt is not paid back all at once, technical debt should be paid back step by step as well. In addition, it cannot be emphasized enough that some of the technical debt does not even need to be paid back at all. Furthermore, as Buschmann [8] pointed it out, paying down technical debt has its own risks as well. Given that many types of debt repayment actions can affect software in production, value can be destroyed as well and not just created. Therefore, it is essential to use minimally invasive solutions.

Development teams can do repayments at micro and macro levels as well. On the one hand, micro level repayments should be continuously done by developers working on the code base. Thus, repayment certainly results to be less burdensome to them. While on the other hand, macro level technical

debt (typically architectural debt) needs to be addressed in recurring clean-up releases.

3.6.2 Means of repayment

Li et al. [25] also addressed the topic of technical debt repayment and alongside with many other authors, also mentioned refactoring, rewriting, automation and fixing of regressions as possible ways of paying technical debt down. However, since all of these are rather focused on the already existing code base, this chapter introduces a few more approaches that can address other types of debt as well, such as the creation of tests, skill management and the revision of communication practices.

3.6.2.1 Refactoring

Refactoring is probably the most commonly known and applied way of technical debt repayment. During the process, the already existing source code is changed in a way that the functionality of the component remains completely intact. However, since the process has no visible outcomes, managers tend to be rather reluctant to the idea of refactoring. It is considered a waste of time, especially if it is carried out with the goal of avoiding future problems that may or may not arise. As a consequence, most of the time people say: *"If it's not broken, don't fix it"*

In general, senior developers are the ones who propose and advocate refactoring, since they see it as an investment. In contrast, junior developers tend to lack the experience and necessary vision to suggest changes of this type. However, regardless of experience level, refactoring is always preferred to be carried out in a progressive fashion. In other words, Uncle Bob's boy scout rule needs to be the main driver of everyday work of developers: *"Always leave the code that you are editing better than you found it."*

3.6.2.2 Rewriting from the ground up

Sometimes the accumulated technical debt reaches extremely high levels in a project. In these cases, instead of fixing all the issues, it is better to abandon the existing source code and re-implement everything again from the ground up. Self-evidently, the fewer lines the affected code contains, the more acceptable this approach is by less technical people as well. On a related note, this repayment method requires rather precise estimates both for fixing the existing technical debt and for the effort of re-implementation.

3.6.2.3 Automation

Manually done processes tend to be repetitive and therefore, incredibly error-prone. Therefore, as indicated in Appendix A, process debt can be partially remedied by automating processes. Some typical examples would be the introduction of Continuous Integration (CI) and Continuous Delivery (CD) solutions.

3.6.2.4 Fixigin regressions

The need for this kind of repayment is the most visible to everybody, since it has to do with broken functionality. Although Li et al. [25] mentioned resolving bugs in general — including defects — as means of repayment, it is a statement that is difficult to agree with in this methodology, since software defects are not considered to be technical debt. However, unnoticed regressions, which are introduced when other bugs are fixed, do form a relevant class of technical debt. Therefore, resolving those issues is also one of the ways of reducing debt.

3.6.2.5 Writing tests

Self-evidently, testing-related debt (see Appendix A) can only be paid back by writing (more) tests. Since software has various requirements, several different types of tests exist. Corresponding to the type of requirements, tests can be divided into two groups: functional (e.g., unit, integration, regression) testing and non-functional (e.g., performance, security, usability) testing.

A good option to consider here is adopting Test-Driven Development (TDD) to eliminate the problem of insufficient testing at its roots: during development time. As part of this software development process, a set of tests are written before the actual code is implemented, which quite self-evidently, initially have to fail. Therefore, the goal is to create an implementation that enables the tests to pass. This helps to avoid the *"I can see that my code works, there is no need for tests. I know I wrote it well."* kind of thoughts that some developers might have.

3.6.2.6 Educating people

As described in Appendix A, there is a type of debt that is related to human resources of projects. One way to repay this debt is by recruiting professionals with just the right skills from the beginning. However, due to different constraint (e.g., urgent need for a developer), this is usually impossible to do in practice. The only viable options are either teaching the necessary

skills to employees or simply finding ways that facilitate their learning on their own. In addition, it is also worth encouraging employees to broaden their professional horizons and not to get stuck doing the same kind of work during years.

3.6.2.7 Revising communication practices

This repayment method aims to address the topic of process and requirement debt, by checking whether the right communication practices are used. For instance, it might be the case that the preferred way of communication is via emails, in order to make sure that managers can monitor the flow of activities by receiving carbon copies of every email. However, many times, more direct, instant messages are more powerful tools. Therefore, the recommendation is to use live communication (e.g., face-to-face meetings, calls and instant messages) for everyday activities and save more permanent means of information exchange (e.g., emails) for making announcements and communicating important decisions. The next chapter provides further details on the communication of technical debt itself.

3.7 Evaluate results and communicate technical debt

This step has a vital role in the overall process, since it helps to determine the success of one technical debt management cycle and it also has the potential to validate the effort that is spent on technical debt reduction and management. Unfortunately, due to the nature of their work, managers tend to think that if projects do not show visible results, time and resources are being wasted. Therefore, providing the stakeholders of projects with numbers about the negative effects of technical debt and the gains of its management is one of the most fundamental principles to be followed.

A good approach to evaluating results is by comparing how much time a certain change required before and after eliminating the technical debt that was linked to it. Paying down debt has the opposite effect on productivity as taking on debt: in the short term it leads to a loss of productivity, but in the long term the organization benefits from it. All this can be measured in terms of both time and money, as well as the change in morale, quality and risks.

In terms of communicating technical debt, it is important to emphasize that choosing the right tools can make a real difference with regards to the

success of communication. Li et al. [25] proposed the usage of dashboards, backlogs and visualizations for this purpose.

- **Technical debt dashboards:** These dashboards exist with the goal of informing stakeholders about the current status of projects. Apart from a list of identified, pending and remedied technical debt items, they can display metrics and visual elements (e.g., charts, graphs) depending on the measurement strategy that the company applies.
- **Backlogs:** They provide a list of every identified debt instance. They serve the purpose of collecting as many details about the accumulated debt as possible.
- **Visualizations:** Powerful visualizations can be created to communicate issues related to architecture and dependencies, to facilitate the comparison of code metrics and to analyze the propagation of effects of technical debt items.

3.8 Take technical debt prevention measures

This section discusses some general guidelines for keeping technical debt levels as low as possible and under control. The goal is to have a technical debt-friendly culture and environment that fosters efficient software development, reducing the need for going into technical debt. This can only be done by leaving the "comfort zone" of the company and adopting new technologies and approaches as well (naturally, assessing them properly beforehand).

3.8.1 People, culture and environment

In order to implement a good technical debt management strategy, the most important thing with respect to people, culture and environment is to spread awareness about the phenomenon. As statistics showed — introduced in section 2.1.1 — the lack of awareness is a significant issue. People experience the effects of technical debt in their everyday work life, yet they know little about the root cause itself. However, given that proper management of these deficiencies is also in their best interest, this should be otherwise. Software development projects with low amount of technical debt have a higher potential of giving the sense of accomplishment to its stakeholders, which is considered to be desirable.

Clients — as the main stakeholders of projects — also need to learn about the dangers of technical debt and providing them with proper educational

material is clearly not an easy task to do. However, at the end of the day, they are the ones who have the final say whether resources are dedicated to technical debt reduction activities or not. If they understand the real implications behind technical debt, their deadline-oriented attitude can be altered towards a more quality-focused one. Additionally, it is also important to make sure that domain experts understand the main technical details of the system and technical experts understand the key aspects of business as well. For instance, sometimes product owners do not understand the technical implications of their newly introduced requirements, since they do not even understand the high-level architecture of the system, thus putting a huge pressure on the development team. Ideally, this kind of situations should be avoided by explaining them the most important technical details of the project (just like developers need to have a basic understanding of the business domain).

Another people-related aspect is the management of skills. As already mentioned, this can be done by systematically hiring people with the right set of competencies or by training existing employees. It is also worth considering the concept of continuous learning, which in the field of software development has a critical role owing to the speed at which technologies are constantly evolving.

In addition, the notion of T-shaped employees is also noteworthy. In project management, this term refers to those people, who have in-depth knowledge in the field of their specializations (the vertical component of the letter T), but also possess cross-domain skills and abilities (the horizontal component of the letter T). The metaphor is used to describe what are the ideal proportions of the two kinds of skills.

3.8.2 Architectural design and source code

Architectural design can be one of the most painful sources of technical debt. Therefore, paying attention to the creation of proper architecture is also a key step in prevention. In order to keep the structure easily modifiable and maintainable, modularity of the architecture needs to be guaranteed. A widely-used approach for that is to implement the so-called microservice architectural pattern that makes sure that the separation of concerns principle is applied. This is done by creating the network of loosely-coupled services, each of which serves a specific purpose. This pattern can be effectively used to replace brittle, monolithic applications that are hard to maintain.

Similarly, another architectural aspect is the management of dependencies. The most important goal is to have every dependency explicitly declared and keep the number of dependencies as limited as possible. The thesis pro-

vides a few more details about this in the description of the second step of the methodology (section 3.2.2).

Furthermore, design patterns should be used everywhere in the source code as well in order to avoid the most common mistakes by design. The usage of these patterns increases the internal quality of code. Furthermore, related to software quality in general, another important aspect is to have reliable testing practices that can validate the software against various different quality concerns, such as bugs, regressions and non-functional requirements.

Another thought to consider is that one should consciously dedicate enough time to the non-functional requirements of software products as well (e.g., their security or proper auditability), since these aspects tend to be taken less seriously due to the pressure of feature development. Depending on the application type, the requirement of security can be addressed by complying with well-known good practices, such as those proposed by the OWASP organization. Concerning auditability, logging has to form an elemental part of software products. However, in case of microservices, it also comes with an extra difficulty: it has to be done in a centralized fashion. Fortunately, this issue can be overcome by using dedicated tools, such as the widespread ELK stack (Elasticsearch, Logstash and Kibana).

3.8.3 Development practices

Everyday practices during software development can have an effect on the success of technical debt reduction and management as well. Essentially, the two most important aspects in terms of practices have to do with technical debt monitoring, repayment and communication.

As per monitoring, the main recommendation is to track every potential or already identified technical debt item. In order to do that effectively, software practitioners — especially developers — need to change their mindsets. Naturally, adding every bit of technical debt to a tracking system manually is often seen as an extra burden, but using a continuous technical debt identification mindset can make it less burdensome. A good way of detecting technical debt at the earliest possible point in time is by doing pair programming and creating pull requests, thus making sure that freshly written source code undergoes some type of code review. It is also important that developers are completely open about their deliberately caused technical debt. However, this is difficult to achieve, since technical debt is often seen in a bad light.

With regard to technical debt repayment, an interesting question is how new technologies are applied at a given company. Is there a knowledgeable senior for every major technology or most of the technologies are learned by

searching the web for tutorials and explanations? Self-evidently, a person with actual experience can better point out all the fundamental principles and best practices of a technology used than just a few Internet searches can.

With reference to communication, clients need to be kept involved in the development loop. Therefore, the usage of issue and project trackers — such as Jira — is highly recommended. But these tools are not the only possible solutions for the problem; for instance, shared Trello boards can also be beneficial in this regard.

3.8.4 The role of Agile development

Adopting the Agile methodology not only comes with its well-known benefits, but it can also help us in devising and implementing better technical debt management strategies as well. Since technical debt in essence refers to the lack of satisfactory quality of any kind of development work, Agile methodologies are the perfect match for its management. Agile focuses on keeping quality at a constantly high level, for instance by making sure that the master branch of a project repository only contains source code which is release-ready at all times.

In pursuance of that, Agile methodologies ship with their own ways and best practices. The list below contains a few of these:

- **Definition of "done":** Although, this definition needs to be refined and redefined according to the needs of the teams in question, in general, work on a backlog item is completed only when the change is ready to be released. Project managers need to take this requirement seriously and keep items from being "done", until they actually meet the criteria.

- **Constant communication:** Due to frequent meetings (e.g., daily stand-up, sprint planning, sprint retrospective) and the usage of visualization tools (e.g., burndown charts), Agile makes sure that the work progress is constantly communicated towards all the relevant stakeholders, thus increasing transparency and the chances of detecting deficiencies early on.

- **Built-in prioritization:** As part of planning meetings, prioritization of tasks is done based on effort estimates before the actual iteration (i.e., sprint) even begins. This helps to address the most urgent tasks first and also make sure that — in an ideal situation — nobody experiences extra workloads all of a sudden, which would force them to start using sub-optimal solutions here and there, thus introducing technical

debt. However, on a related note, Nord et al. [32] indicated that story points usually do not include the time required for addressing potential technical debt while working on a task. Therefore, estimates sometimes do not capture the real impact of technical debt and this also needs to be taken into account during the planning process.

- **Short feedback cycles:** Due to efficient communication practices and short iterations, software development teams get customer feedback early on, which keeps the amount of requirement debt under control.

Although, Agile is a powerful methodology, sometimes it is impossible to adopt it on account of various possible reasons. One such obstacle can be the reluctance of clients to cooperate in an Agile way. Luckily, it does not mean that Agile is of no help in these cases: some Agile tools and concepts can be still utilized (e.g., peer programming, peer reviews, internal meetings), but clearly with a reduced level of efficiency.

Chapter 4

Case study

This chapter describes how some elements of the presented methodology can be used in practice, at a software development company.

During my internship at a small software development company, I had the opportunity to run a simple case study in order to test the practical usage of the proposed methodology. However, due to the short duration of the internship, I only had the opportunity to examine one iteration of the technical debt reduction methodology.

To put the case study in context, it is worth mentioning that the company was founded slightly more than two decades ago and it currently has about 20 employees. Its main vision is to create solutions that can be used for the extraction of actionable data from various types of unstructured data sources via text analytics. These activities — among others — consist of social media analytics, customer feedback analytics and employee feedback analysis. As an additional activity, the company also does data analytics on already extracted data.

For this case study in particular, I decided to consider Java Spring Boot projects that are related to the most important client (a big company from the healthcare and pharmaceutical domain). Initially, the client of the company contracted them with the development of an application for only one market. However, later on, their request was extended to several different markets and applications with very similar functionalities. Unfortunately, the initial project was not developed keeping evolvability and maintainability in mind.

The already existing projects contained a large amount of technical debt, which is due to the high number of innovation projects that are involved in the development process. As it has been already indicated earlier, these kinds of projects are always more prone to accruing technical debt than others. Therefore, it was important to start the explicit management of technical

debt now, since only a few more markets have been implemented so far.

4.1 Understanding the company environment

Since the goal is to collect as much information as possible, the larger number of approaches we use, the better the results are expected to be. Therefore, I identified the following methods as viable options:

- **Interviews:** This method refers to those live conversations, which are carried out with employees of different roles. For example:
  - *Product owners and product managers* can help to better understand business aspects.
  - *Scrum masters and people with similar Agile roles* can explain the role of the Agile methodology at the SME.
  - *Software engineers* can shed some light on technical details of projects.
- **Source code:** Reading and understanding the source code, which helps with technical aspects.
- **Documentation:** Although this option is not necessarily available, it is supposed to serve exactly the same purpose as this step.
- **Visualizations:** This refers to visual representation of the whole or parts of the system architecture, which helps to understand the relationship between different components of it. However, this is another one of those approaches that might not be viable in every case.

The table below compares these four approaches in terms of the time needed for information extraction, the information gain and their availability. Taken every aspect into account, interviews are found to be the most optimal alternative.

Method	**Time**	**Information gain**	**Availability**
Interviews	Medium	Medium	Guaranteed
Documentation	Long	High	Not guaranteed
Source code	Long	High	Guaranteed
Visualizations	Short	Low (restricted to some aspects)	Not guaranteed

Based on the results of the comparison, I organized several informal interviews that were all carried out in person, since everybody works at the same office building. The subjects of these were not only the CTO of the company, but also other employees. Depending on the topic of these conversations, their duration ranged from short (5 minutes) chats to longer (1 hour - 1.5 hours) discussions. It is noteworthy that these interviews could only yield good results, because the team had a global understanding about ongoing projects. Unfortunately, documentation of the chosen projects and the overall architecture was not available with the exception of occasional comments in the code base. Therefore, I identified some documentation debt at the first step already.

This step provided a good basis for the rest of the methodology. First of all, the main conclusion was that — due to their innovative nature — the selected projects accumulated plenty of technical debt (almost every type that was identified in Appendix A). This could happen owing to the fact that at the moment of implementing the initial solution, it was not prepared for this kind of future scenarios. Secondly, I identified some missing best practices and cultural aspects (e.g., almost complete lack of tests, not being Agile) that favor accruing technical debt. Thirdly, I understood that the product greatly relies on the Cloud, which has its own implications. Fourthly, I also found that clients of the company differ from each other in terms of the ease of cooperation and communication with them. Although, there are clients who manage these things professionally (e.g., they are somewhat agile, they use good communication practices), others are rather messy, thus promoting the accumulation of technical debt in the corresponding projects. The client that I chose for this case study in particular also belongs to the latter group. They try to have a micromanagement approach to a set of technical projects, but they lack the necessary technical skills to properly do so. Consequently, communication with this client is also cumbersome.

4.2 Technical debt identification

In order to identify technical debt at the company, I used three approaches. While issues of the architecture were **identified by experts** and **by dependency analysis tools**, source code-related problems were discovered by **static code analysis (SonarQube)**. The following sections examine alternatives with respect to each approach and also present some of the technical debt items that were identified and remedied during the case study.

4.2.1 Static code analysis

There are several tools available for different technologies and programming languages, but this section introduces the three most referenced ones. Although, I found that all three tools can be used for various technical debt related activities, such as identification, measurement, monitoring and prioritization, this document introduces them here, since in principle, they are all static code analysis tools.

- **SonarSource suite:** It is a continuous code quality toolset which consists of SonarQube, SonarCloud and SonarLint. One of its main advantages is that it supports various programming languages, CI engines (e.g., Jenkins, Azure DevOps, Travis CI) and build systems (e.g., Maven, Gradle, MSBuild, Ant). With respect to issue identification, these tools can detect bugs, code smells, security vulnerabilities and low test coverage as well. In addition to that, all of these features are built on customizable rule sets where each rule can be separately enabled/disabled according to the needs of the team. Finally, the SonarQube dashboard also provides visualizations of metrics, quality ratings and quality gates (to enforce certain code quality for the code that is released).

- **SQUORE Software Analytics:** It is a software intelligence tool for project quality and performance. It uses different kinds of artifacts (e.g., source code, test results, bug tracking systems, output of other tools) to aggregate data with its own results and display a summarized view of the project. Every artifact is publisherrated (using the SQALE method) and trends can be analyzed using visualizations. The role-based SQUORE dashboard helps evaluating technical debt in terms of list of instances, key performance indicators, the overall trend, its distribution per module, the SQALE pyramid, violation density, complexity debt and cloning debt. A huge advantage of the dashboard is that it uses various types of visualizations as well as in-depth statistical data.

- **Kiuwan Code Analysis:** The main goal of this tool is the detection of defects as part of the contiuous development process, with a customizable set of rules. It aims to address risks, security, maintainability, reliability and efficiency of applications. It can prioritize the effort of managing technical debt automatically and create action plans based on the goals of the team or the available resources. The evolution of these plans can be also monitored by Kiuwan. It provides support for

all the major languages, IDEs, build systems, bug tracking systems and version control systems. Additionally, it also has the ability to integrate with other analyzers (e.g., PMD, Findbugs and Ckeckstyle among others). By integrating with IDEs, code is analyized every time it is saved and the editor is decorated with defects accordingly to help the work of developers.

The table below compares these three tools with respect to supported technologies, IDEs, execution environment and the open-source license.

	SonarSource S.	**SQUORE S.**	**Kiuwan C. A.**
Technologies	ABAP, Apex, C, C++, Objective-C, COBOL, C#, CSS, Flex, Go, HTML, Java, JS, Kotlin, PHP, PLI, PL/SQL, Python, RPG, Ruby, Scala, Swift, TS, TSQL, VB, XML	Ada, C, C++, C#, Java, Cobol, PL/SQL, ABAP, PHP, Python	ABAP, AS, ASP.NET, C, C++, C#, COBOL, HTML, Informix, Java, JS/TS, JCL, JSP, Natural, Objective-C, OracleForms, PHP, PL/SQL, PS, Python, RPG, Swift, TSQL, VB, Groovy, SQL, Scala, Ruby, XML
IDE support	Eclipse, IntelliJ IDEA, Visual Studio, VS Code	IDEs are not supported	Eclipse, IntelliJ IDEA, Visual Studio
Where?	On-premises or using their Cloud-based service	On-premises	On-premises or using their Cloud-based service
Open-source?	Yes	No	No

Having compared these tools, I decided to use SonarQube for static code analysis, since it is an open-source tool (consequently, it is free to use) and it

offers a rich set of features. I installed it on one of the servers and configured the CI pipeline to run the analyses of the projects every time new code is pushed to a given repositories. However, it is worth mentioning that the results of the analysis do not affect the execution of the pipeline (i.e., it is never aborted due to failing a quality gate). I also made some adjustments to the default settings, such as disabled some of the rules that the development team did not deem necessary. Although the results of the analyses were overwhelming at first, with the right attitude they proved to be extremely beneficial. In order to showcase its features and how it helped the company, I chose a rather new project (approximately 2 months old), since it was started during my internship and its evolution provided me with good measurements and results.

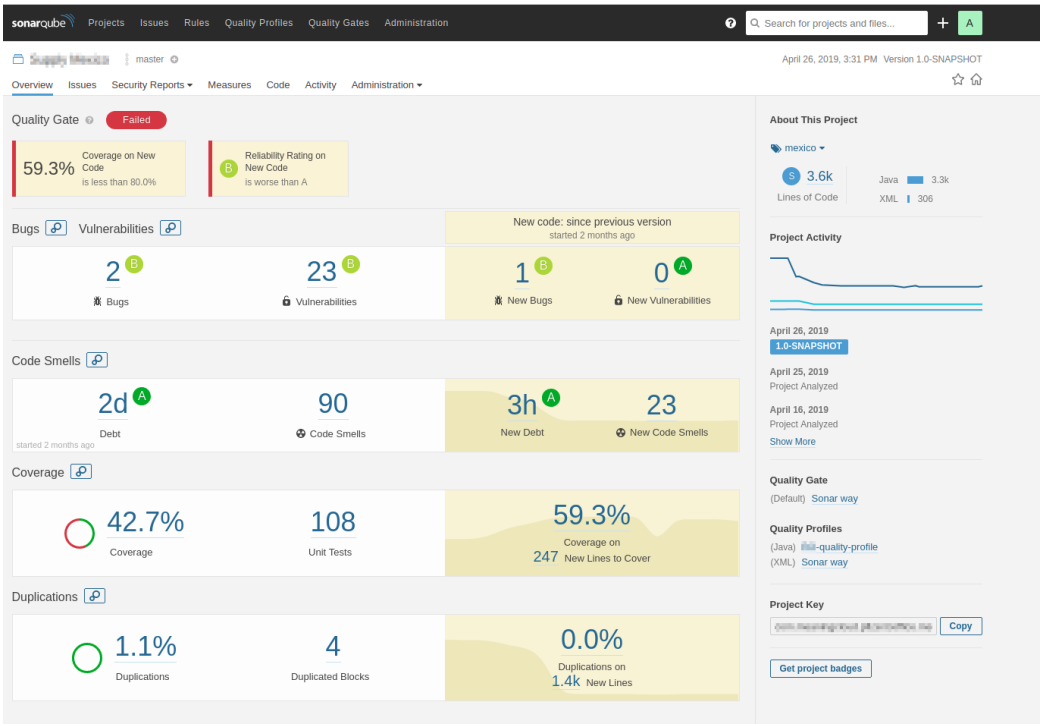

Figure 4.1: Technical debt identification - SonarQube dashboard - Overview

As it can be seen in Figure 4.1 above, SonarQube identified 2 bugs, 23 vulnerabilities and 90 code smells in the aforementioned project. The platform also displays the estimated amount of technical debt, which is fortunately rather low in this case, since the project was developed taking technical debt-related recommendations into account. It is also worth mentioning that the code coverage is well-above average in this project — although, it is still not at a satisfactory level. In addition, the number of duplicated lines is low,

which is also an indicator of better code quality. In general, it can be said that these numbers are not as low for every project as for this one. However, they are slowly, but surely improving due to the measures taken against software deficiencies.

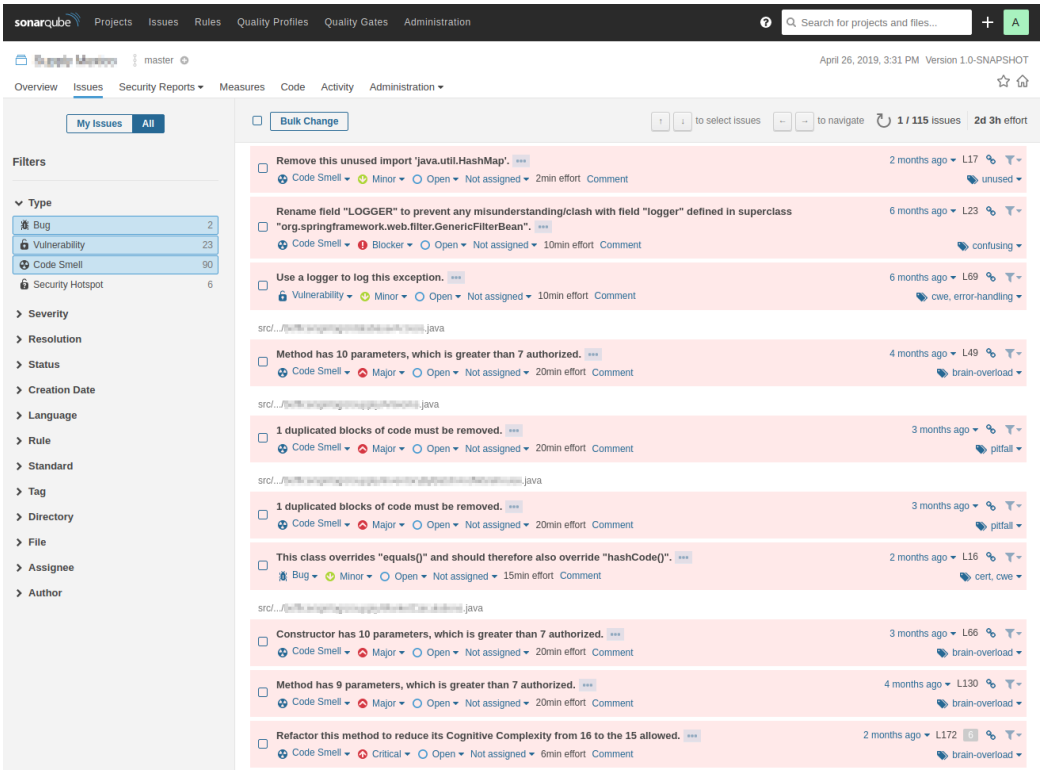

Figure 4.2: Technical debt identification - Issues

Figure 4.2 shows the detailed list of issues. By clicking on an item, the corresponding part of the source code appears, alongside with a more in-depth description of the issue, which helps to understand why a given issue is dangerous. In order to further aid decision making, all of these issues can be filtered using various types of filtering criteria. Although, it is impossible to address all the 125 issues that were identified in this document, the analyzer found important problems such as poor exception handling (e.g., when only the stack trace was printed to the standard output, but events were not logged by any loggers).

4.2.2 Dependency analysis

I also examined what my options were with respect to carrying out a dependency analysis of the projects. In general, I found that there was a lack of

universal tools for this purpose due to the industrial tendency of introducing a new build system or dependency management system for every new language. Some of these tools and solutions are listed below:

- **Java:** Maven, Gradle
- **Python:** Pipenv
- **NodeJS:** Yarn, NPM
- **PHP:** Composer
- **Objective-C/Swift:** CocoaPods

One thing that is common for all of these tools is that they provide a way of visualizing the dependency graph/tree and most of them can also run an automated dependency analysis. Furthermore, IDEs also help with the discovery of dependency-related issues. For example, IntelliJ supports the detection of backward dependencies, cyclic dependencies and module dependencies, but Eclipse and Visual Studio are also equally well-equipped to aid dependency management.

At the case study company, every project used Maven initially, but it started to make the development process overly complicated as the number of dependencies on in-house projects grew and it also made it impossible to manage dependencies of projects separately. Hence I replaced it with Gradle during the technical debt reduction process. I had to improve the dependencies of the projects with the goal of escaping from the so-called "dependency hell", which was already taking its victims.

Understanding the hierarchy of dependencies is often a strenuous task to do, however, my work was supported by the following two features:

- **Dependency trees:** It displays the hierarchy of dependencies as a tree.
  `mvn dependency:tree`
  `gradle module-name:dependencies`
- **Dependency analysis:** It finds "Used undeclared dependencies" and "Unused declared dependencies"
  `mvn dependency:analyze`

Using the above mentioned methods, for instance I could replace transitive dependencies with explicit dependencies and also remove unnecessarily declared ones (e.g., instead of using the entire AWS SDK, use just the modules that were actually needed).

4.2.3 Identification by experts

Experts of the company also identified several — mostly architectural — issues. These technical debt instances are listed below.

- I found that due to the lack of initial design, the application had become the typical case of a "big ball of mud". It was **a monolith with serious architectural problems** that needed to be addressed.
- **Abstractions were missing.** For instance, instead of using a generic storage class, the code kept referring to files on the hard drive, which should be avoided in a Cloud environment.
- **The logging of the application was not centralized.**
- There were manual **processes that could be easily automated**.
- **Maven was used as a build system, which made the build and development process rather complicated.** In case of starting work on a new project, in order to build and run it, first its dependencies also had to be installed, which required a lot of manual effort when other in-house projects were involved as well.
- Every time a developers started working on a new project, besides importing the source code of it, they also had to provide a set of configuration files. Therefore, the **configuration management had to be improved** as well.

4.3 Technical debt measurement

On the one hand, I decided to rely on SonarQube with the measurement of source code-related technical debt. On the other hand, architectural technical debt was measured manually.

As already mentioned, SonarQube counts the number of bugs (reliability rating), vulnerabilities (security rating) and code smells (maintainability rating). Additionally, it also provides the following metrics: test coverage, code duplication, size (e.g., number of lines of code, number of classes), complexity (e.g., cyclomatic complexity) and statistics related to issues. Furthermore, as Figure 4.3 shows, the analysis of these aspects is also supported by bubble charts. For instance, using the chart below, it is rather easy to see that the selected project has a large number of code lines with "A" reliability and security rating, having a test coverage around 80% and an average of 2 hours 30 minutes worth of technical debt.

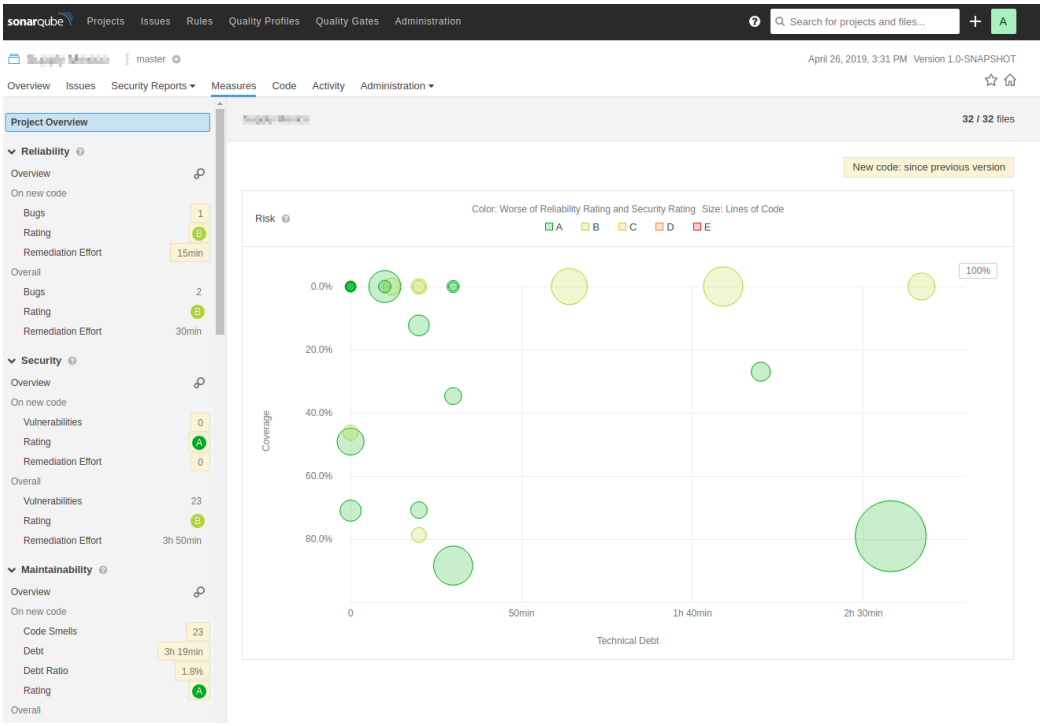

Figure 4.3: Technical debt measurement

As per the architectural debt, it is only measured in retrospect. This is done manually, using the spreadsheet that will be introduced in the next section, which discusses technical debt monitoring.

4.4 Technical debt monitoring

The case study company monitors technical debt in three different ways. First of all, SonarQube is used to analyze the trends of each project. Secondly, there is a manually maintained spreadsheet to track those types of debt that SonarQube cannot identify. Lastly, developers occasionally also use "TODO" comments in the source code. However, they try not to do so if possible, since they are considered bad practice. No backlogs or issue tracking tools are used for this purpose.

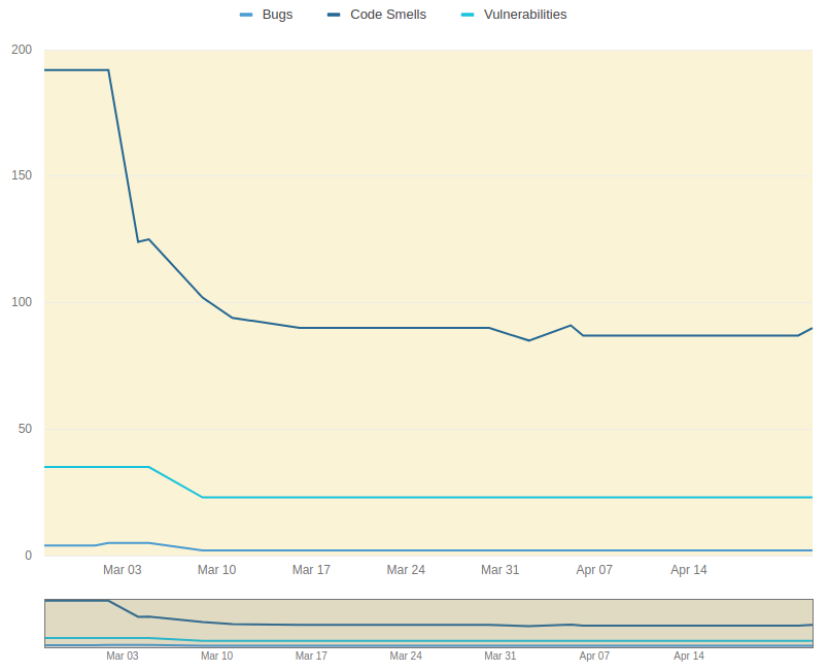

Figure 4.4: Technical debt monitoring - Issues

SonarQube provides powerful visualization about activities related to projects. Figures 4.4, 4.5 and 4.6 depict the three main types of predefined monitoring graphs: issues, test coverage and code duplications respectively. As it can be observed, owing to to the new technical debt management measures, the number of bugs, code smells, vulnerabilities and duplicated lines of code decreased. Additionally, the test coverage of the project also underwent a positive change, since it increased. However, it did not reach satisfactory levels.

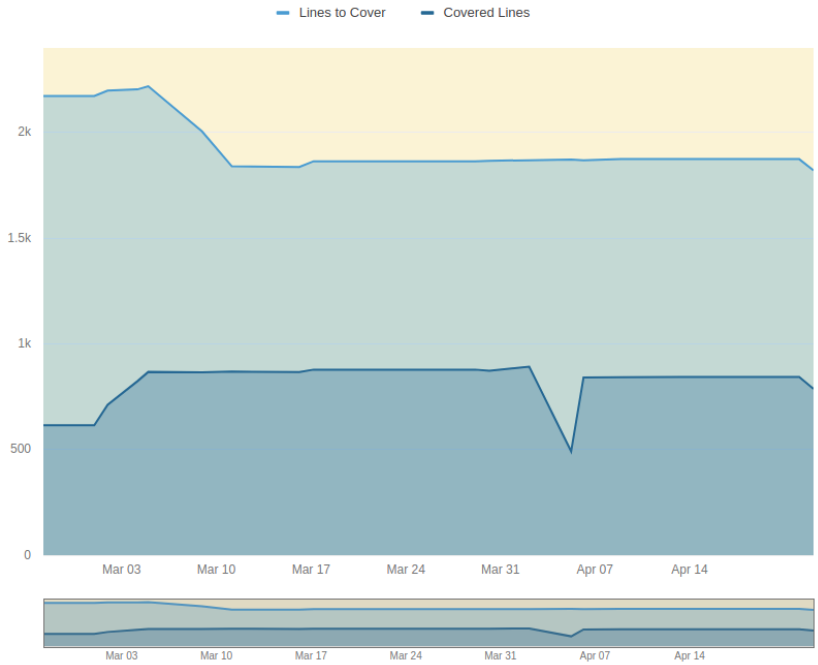

Figure 4.5: Technical debt monitoring - Test coverage

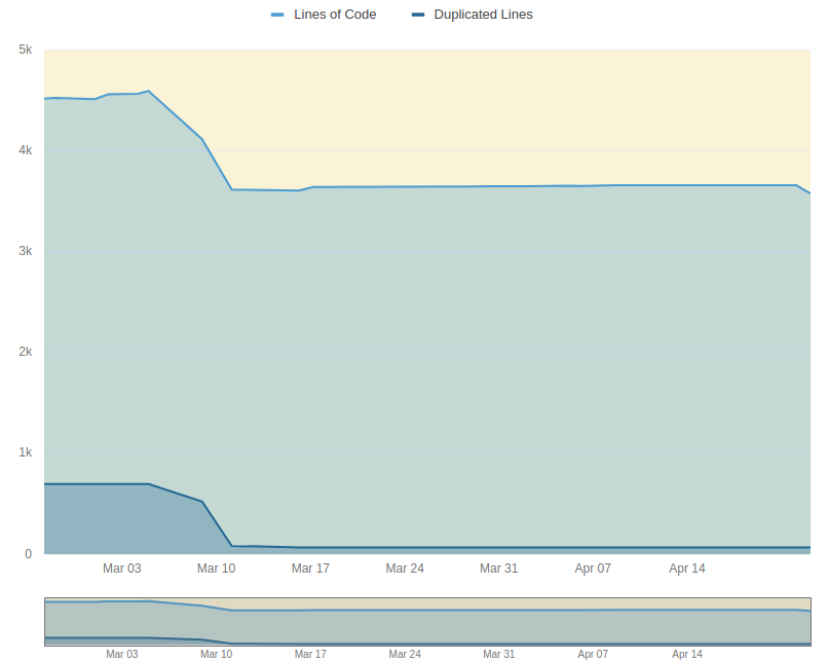

Figure 4.6: Technical debt monitoring - Duplications

As per monitoring architectural debt, it is based on the technical debt spreadsheet. It contains the following information about each identified architectural technical debt item:

- An **ID** that identifies each item without ambiguity.
- **IDs of related items** (if any).
- **Date** of registering an item.
- **Type** of the registered item.
- Name of the **responsible person**.
- **Description of the change** to be made.
- **Time required to do the change** in a completely ideal situation.
- **Description of the additional changes** needed (paying down the principal).
- **Time required to do the refactoring** of the corresponding components or the **reengineering** of aspects.
- **The impact** of certain technical debt instance, measured by the **percentage of affected projects**.
- **The risks of destroying value** by paying down the technical debt item (low, medium, high).

4.5 Technical debt prioritization

Since the company did not have a technical debt management strategy before the beginning of the case study, my job was to implement a completely new strategy for this purpose. A crucial element of creating such a strategy is choosing the means of technical debt prioritization. At the internship company, I introduced a two-level prioritization strategy, combining the "highest remediation cost first" and the "highest interests first" strategies.

The first level prioritization is simply done by grouping debt instances into two groups: architectural debt and non-architectural debt. Out of these two, the former one has a higher priority, because architectural changes might render some of the remaining technical debt irrelevant. Therefore, the second level of prioritization is also applied to that category first.

At the second level, always the most burning issues are addressed. Therefore, as already explained in section 3.5.2.3, this approach focuses on the repayment of those technical debt items first that demand the highest (and most frequent) interest payments. This strategy is also equivalent to the *"If it's not broken, don't fix it"* attitude. It is also noteworthy that SonarQube provides powerful filtering mechanisms to examine non-architectural debt with the goal of determining the necessary details.

4.6 Technical debt repayment

During the case study, the company paid back a large amount of technical debt. Therefore, the following subsections aim to present some of the most significant changes that my colleagues and I made.

4.6.1 Splitting the monolith

As already mentioned in the technical debt identification step (section 4.2.3), the application used to lack modularity and the separation of concerns principle was not applied properly. What made the situation even worse was the lack of tests, since in such scenarios, regressions are quite often introduced and left unnoticed.

In order to remedy this situation, with the help of my colleagues, I implemented a microservice architecture by creating a network of loosely-coupled services, where each of these services have a well-defined role and functionality within the architecture. As a consequence, the company gained control over individual components and also decreased the number of newly introduced regressions significantly. The whole refactoring work took them approximately 32 hours (4 entire workdays), however, it also saved them all the extra time that they would have to spend on constantly fixing regressions in the future for example.

4.6.2 Introducing abstraction levels

The initial architecture did not have abstraction levels designed properly. As already mentioned, in the introduction of this chapter, different markets require very similar features. Consequently, the source code of newer markets build on that of older ones. Therefore, it was essential to introduce abstraction levels now, before the number of markets further grew.

One good example of this is how I replaced every reference to files on hard drives with a storage abstraction interface. As a result of this, the means

of storage can be changed at any moment (using the configurations of the application only) without the need of changing the source code at several places. The refactoring activity took me 24 hours, but it made it possible to implement new storage types in a very short time. For instance, the S3 storage was introduced in 4 hours, but it would have cost much more without the abstraction layer.

4.6.3 Making logging centralized

With respect to logging, there were two aspects that needed to be addressed. One of them was the list of issues that were identified by SonarQube (e.g., only printing the stack trace to the standard output) and the other one is the implementation of a solution that would enable centralized log management. While the former could be easily fixed by refactoring the corresponding parts of the source code, the latter required more of an effort.

After researching the topic, I found that the best available option was to use the ELK stack, which consists of three main components: Elasticsearch, Logstash and Kibana. While Elasticsearch is a powerful search engine used for logs, Logstash and Kibana are just as impressive ingest pipeline and visualization tool respectively. Setting up and configuring all these components made it possible to manage the produced logs of every individual microservice centrally.

4.6.4 Automating processes

As stated earlier, I identified some automatable processes as well. The most important one of these was the deployment process, since its automation could save a significant amount of time. The first version of the process — from the time before the monolith was split — is displayed in Figure 4.7. Systemctl used to run a “run.sh” script, which basically used a single JAR file (including all the configuration files) to start an application instance. However, this process did not include CI/CD and the configuration management was poor as well. Therefore, it had to be changed.

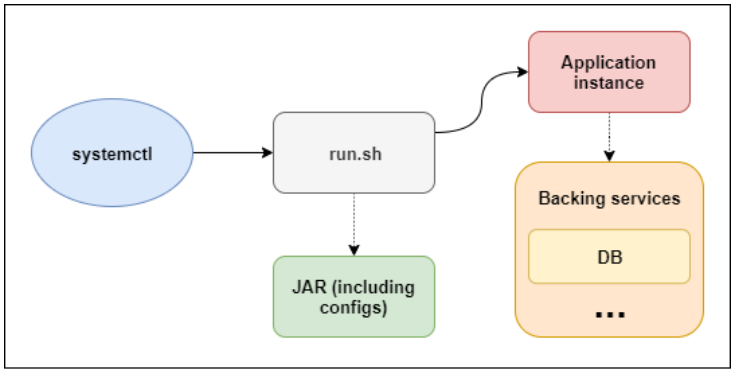

Figure 4.7: Technical debt repayment - Process v1

Once the microservice architecture was introduced, I also added the well-known automation server, Jenkins, thus making a step towards CI/CD as well. Naturally, this also had to involve paying down some people debt and motivating/educating developers to write tests. The internalization of test-driven development is still an ongoing process. In version 2, components of the applications were also split into several JAR files and configuration files were separated from the source code to increase maintainability. Load balancing was done based on a spreadsheet containing a list of hosts and ports.

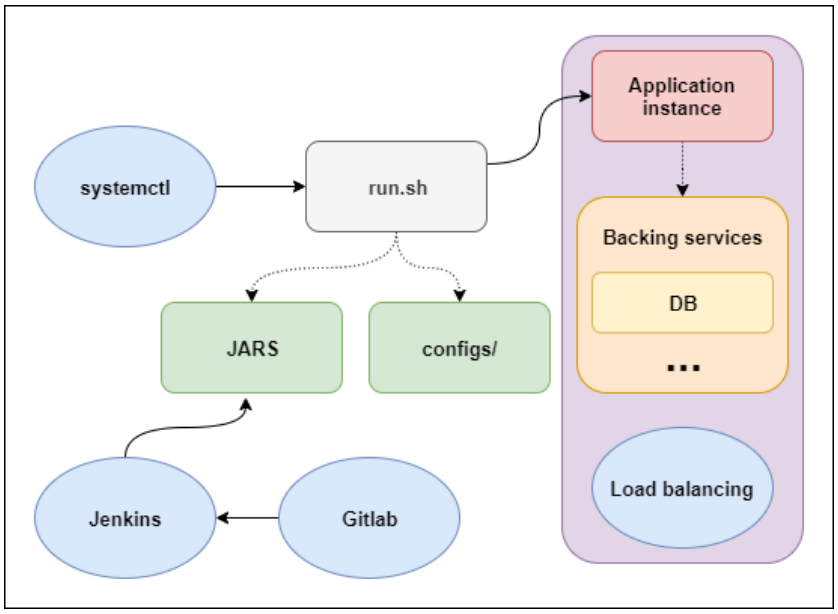

Figure 4.8: Technical debt repayment - Process v2

In the final version, Systemctl was replaced with Nomad as a scheduler. In this case, Jenkins is responsible for compiling the JARs and uploading them to Amazon S3, generating Nomad templates and invoking Nomad as well. Next, Nomad downloads a given JAR from S3, runs it on some machine and it also registers the IP and port number of the machine in Consul, the service discovery server.

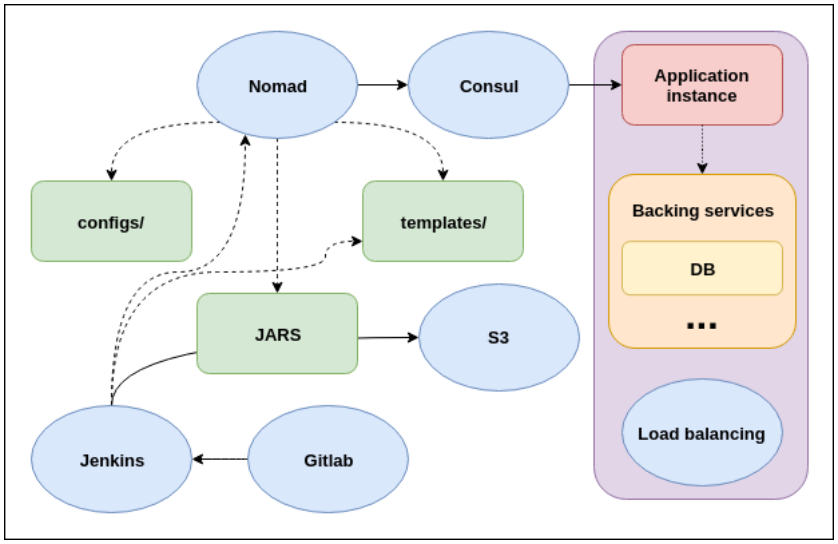

Figure 4.9: Technical debt repayment - Process v3

Additionally, I also automated the setup of the development environment with the help of Gradle. In accordance with what was explained in section 4.2.3, while using Maven, developers had to manually clone and install the internal dependencies of an application first (using the "pom.xml" files shown in Figure 4.10) in order to be able to build it. In addition, the maintainability of such a large number of "pom.xml" files also proved to be very low.

In the new Gradle structure, I introduced a "repos.gradle" file, which contains the details of the repositories that are used by an application. Using Gradle scripts, I automated the cloning steps of the listed components into the "lib" folder. Furthermore, Gradle also takes care of their compilation. By doing this, developers only need to open the application project and everything else is automatically downloaded and configured now.

4.6.5 Reengineering

I also did some reengineering work. I realized that storing credentials of certain services (e.g., Amazon AWS) in the source code was to be considered

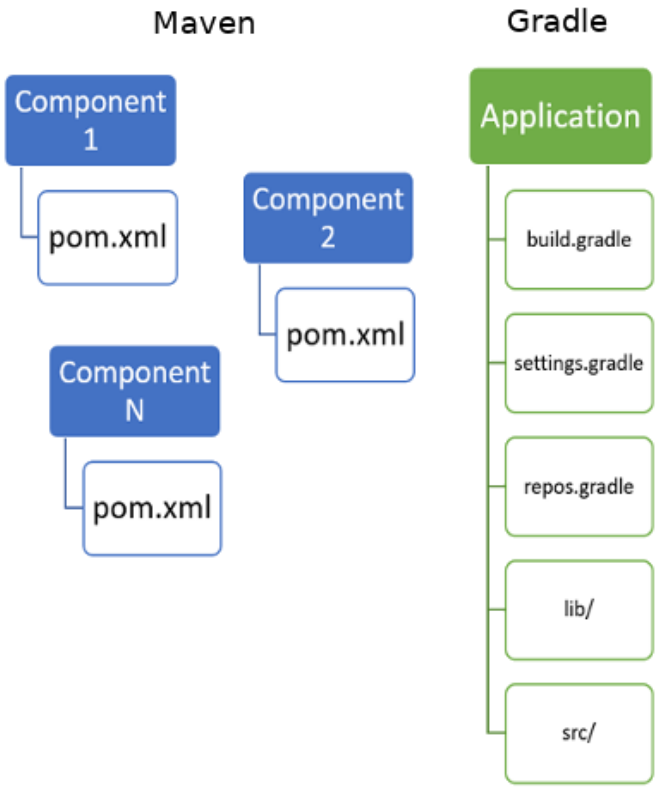

Figure 4.10: Technical debt repayment - Build tools

a security vulnerability, since they were also committed to Gitlab for instance.

Therefore, I looked for alternatives and found a product called Vault, developed by HashiCorp. This tool provides the means of managing different kinds of secrets (e.g., passwords, tokens) of the applications in a centralized and secure way, in cooperation with the previously introduced Nomad.

4.6.6 Documentation

Documentation debt is hard to repay in retrospect, since there is no budget available for such activities in general. Therefore, the only thing I could do about it is trying to document newly developed pieces of software better. Fortunately, some of the developers already made a huge effort to do so.

However, it is also worth mentioning that the need of documentation was argued at the company, since any kind of documentation — e.g., textual documents, visualizations, wiki pages and code comments — need to be kept up to date, which is often impossible to guarantee. Therefore, in addition

to the explicit documentation, developers of the company aim to keep the source code as clean as possible, so that it can serve as documentation itself.

4.7 Technical debt evaluation and communication

A few months ago, the "external" client of the company was not very open to technical debt-related discussions. However, their attitude towards the matter has significantly changed. This could only happen by being persistent in communicating technical debt repayment needs and by also involving them in the development process as much as possible. Slowly, but surely they understood the importance of technical debt management, by seeing the benefits of its repayment. As a result of that, they also accepted the fact that sometimes, seemingly simple changes require large refactoring efforts.

However, they rejected the efforts of the company to share and discuss the SonarQube dashboard with them, since they have a preference to focus on burning technical debt issues only. On the other hand, this dashboard and the architectural technical debt spreadsheet should still be evaluated and discussed with internal management of the company, considering that they are the "internal" clients.

4.8 Technical debt prevention

Luckily, many of the identified prevention measures are already implemented. However, there are a few more aspects that could be improved, such as communication practices, skill management and Agile practices. In general, prevention is an attitude, which is still in the making at the case study company.

The company needs to further improve their communication practices. Although, there have been some compelling improvements in the communication between the company and the external client already, there is still plenty of room left for increasing the efficacy of communication. One of the huge achievements of the company is that they managed to persuade them to communicate the development progress using Kanban boards in Trello, instead of relying on emails only for every type of communication. They also tried introducing behavior-driven development, to avoid requirement debt as much as possible, however the "external" client was not willing to cooperate. Additionally, they should also make sure that managers do not hinder

communication between technical employees due to their micromanagement approach.

Skill management should also get a high priority. Besides trying to have a wise hiring strategy, the organization should also encourage continuous learning and challenge ourselves every now and then. This is partially already done by sharing links to interesting and useful technical materials with each other, using a dedicated Slack channel, but they could also organize technical training activities. On a related note, most of the developers are currently in the process of adopting the Test-Driven Development approach to lower the accumulation rate of testing debt. However, many of them still need to improve their test writing skills.

Becoming Agile would help the company in managing technical debt. However, it is a goal that is impossible to reach due to the reluctance of the "external" client. Either way, another important task at this point is to implement as many Agile tools as possible. One such example is the increased transparency within the development team, owing to the recently introduced weekly meetings.

Chapter 5

Evaluation

This chapter examines and evaluates the outcomes of the master's thesis project. It discusses to what extent each goal was reached and also presents a few ideas for future development.

5.1 Goals achieved

This master's thesis has successfully accomplished all the originally defined goals. The key achievements of this project are the following ones:

- It provided an exhaustive literature review, thus creating the basis of discussing the methodology.
- It produced a comprehensive methodology that was proven to reduce technical debt.
- It described how easily the methodology could be applied in industrial settings.
- It improved the performance of the case study company.

As per the goal of raising awareness about the phenomenon of technical debt, this master's thesis includes a thorough overview. In order to serve SMEs best, it is easily understandable and only focuses on those characteristics of technical debt that have practical implications. Only by reading the literature review chapter, IT practitioners can gain useful insights that can change their ways of thinking about software development, since rather simple changes can have a positive effect too. It is also worth mentioning that due to the academic nature of master's theses, the author tried to concentrate on academic literature as resources. However, a significant amount

of useful information about the topic can be collected from blog posts and forums as well.

Regarding the methodology, we can conclude that any SME could benefit from it in order to implement a technical debt reduction strategy. Although, it serves as a powerful starting point, it does not cover every single alternative for each step due to the scope restrictions of the master's thesis. Therefore, there might be cases when certain steps or even tools need to be customized for the company. Furthermore, it is also a fact that the role of efficient communication practices is not emphasized enough.

Despite the limitations of the case study (e.g., short duration, involving only one small company), it proved that the methodology can greatly contribute to the implementation of an efficient technical debt reduction strategy. Therefore, both the company and the case study client were satisfied with the methodology and the changes that I managed to achieve. Owing to the implemented incremental change strategy and a team with improving competences (both in terms of tools and languages), I made numerous successful technical debt reduction steps. Just to mention one, a particularly well-received improvement was the introduction of SonarQube. Because of its success, the company decided to use the tool in all of its future projects as well. To conclude, the CTO of the company drew the following overall conclusion: *"We moved from ambiguity and a general sense of 'unknowns' to a controlled state."*.

5.2 Future work

Although the thesis project fulfilled all the goals that had been previously defined, the associated research has also raised some new questions and tasks in need of additional investigation. First of all, the methodology needs to be tested in other companies as well, also including medium-sized enterprises. These tests would be necessary to further refine its steps, in order to make sure that it is also applicable in different environments. Secondly, further examination of people-related aspects and good communication practices is also considered to be very significant. Lastly, the use of artificial intelligence (AI) in this field is not very emphasized yet. Therefore, it would be advantageous to study how AI could make the management of technical debt even more efficient.

Chapter 6

Conclusions

In summary, this master's thesis addressed the problem of technical debt by defining a software engineering methodology that can serve as a good starting point for implementing a technical debt reduction strategy at SMEs. As it can be seen from the previous chapters, it is of key importance to pay sufficient attention to the issues behind the metaphor, since the success of software development greatly depends on the levels of technical debt that a given project has. When it is not managed properly, technical debt can easily get out of control, thus destroying employee morale, reducing productivity, decreasing software quality and above all, increasing risks significantly.

In order to aid the understanding of the phenomenon behind the metaphor, this document also examined several different characteristics of technical debt, such as its definition, its main properties, its different categories, its predominant sources and its most important effects on projects. This literature review was followed by the core component of the thesis: an iterative technical debt reduction methodology consisting of eight steps. Finally, the document also presented the outcomes of the case study, which I carried out at a small software development company.

To conclude, based on the positive feedback and satisfaction of the case study company, it is safe to say that this new methodology successfully served its purpose. By the end of the case study, the technical debt levels of the company were notably reduced, which also lead to a smoother developer experience in general. However, in all fairness, the methodology still needs to be tested in larger organizations as well. Furthermore, as detailed in the previous chapter, there is still plenty of room left for future work in this field. Despite all that, it would appear that the contents of this master's thesis will enable interested SMEs and IT practitioners to successfully reduce their technical debt levels.

Bibliography

[1] Allman, E. Managing technical debt. *Queue 10*, 3 (Mar. 2012), 10:10–10:17.

[2] Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C. Identification and management of technical debt. *Inf. Softw. Technol. 70*, C (Feb. 2016), 100–121.

[3] Alves, N. S. R., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. Towards an ontology of terms on technical debt. In *2014 Sixth International Workshop on Managing Technical Debt* (Sep. 2014), pp. 1–7.

[4] Bohnet, J., and Döllner, J. Monitoring code quality and development activity by software maps. In *Proceedings of the 2Nd Workshop on Managing Technical Debt* (New York, NY, USA, 2011), MTD '11, ACM, pp. 9–16.

[5] Brazier, T. ACCU - Professionalism in Programming - Managing Technical Debt. `https://accu.org/index.php/journals/1301`, Feb. 2007. Accessed March 28, 2019.

[6] Brooks, Jr., F. P. *The Mythical Man-month (Anniversary Ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[7] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I., Sangwan, R., Seaman, C., Sullivan, K., and Zazworka, N. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research* (New York, NY, USA, 2010), FoSER '10, ACM, pp. 47–52.

[8] Buschmann, F. To pay or not to pay technical debt. *IEEE Software 28*, 6 (Nov 2011), 29–31.

[9] Cunningham, W. The wycash portfolio management system. *ACM SIGPLAN OOPS Messenger 4*, 2 (1993), 29–30.

[10] Curtis, B., Sappidi, J., and Szynkarski, A. Estimating the size, cost, and types of technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Piscataway, NJ, USA, 2012), MTD '12, IEEE Press, pp. 49–53.

[11] Ernst, N. A. On the role of requirements in understanding and managing technical debt. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Piscataway, NJ, USA, 2012), MTD '12, IEEE Press, pp. 61–64.

[12] Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., and Gorton, I. Measure it? manage it? ignore it? software practitioners and technical debt. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (New York, NY, USA, 2015), ESEC/FSE 2015, ACM, pp. 50–60.

[13] Finley, E. Evidence defeats doubt: Tips for expressing your opinion. `http://www.dalecarnegiewayindy.com/2011/03/24/evidence-defeats-doubt-tips-for-expressing-your-opinion/`, Mar. 2011. Accessed April 07, 2019.

[14] Fowler, M. TechnicalDebt. `https://martinfowler.com/bliki/TechnicalDebt.html`. Accessed March 14, 2019.

[15] Fowler, M. TechnicalDebtQuadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`. Accessed March 17, 2019.

[16] Greening, D. R. Release duration and enterprise agility. In *2013 46th Hawaii International Conference on System Sciences* (Jan 2013), pp. 4835–4841.

[17] Guo, Y., Seaman, C., Gomes, R., Cavalcanti, A., Tonin, G., Da Silva, F. Q. B., Santos, A. L. M., and Siebra, C. Tracking technical debt ? an exploratory case study. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)* (Sep. 2011), pp. 528–531.

[18] Hilton, R. When to work on technical debt. `https://www.nomachetejuggling.com/2011/07/22/when-to-work-on-technical-debt/`, July 2011.

[19] Holvitie, J., Leppänen, V., and Hyrynsalmi, S. Technical debt and the effect of agile software development practices on it - an industry practitioner survey. In *2014 Sixth International Workshop on Managing Technical Debt* (Sep. 2014), pp. 35–42.

[20] Kruchten, P., Nord, R., Ozkaya, I., and Falessi, D. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *ACM SIGSOFT Software Engineering Notes 38* (08 2013), 51–54.

[21] Kruchten, P., Nord, R. L., and Ozkaya, I. Technical debt: From metaphor to theory and practice. *IEEE Software 29*, 6 (Nov 2012), 18–21.

[22] Laribee, D. Code Cleanup: Using Agile Techniques to Pay Back Technical Debt. `https://msdn.microsoft.com/en-us/magazine/ee819135.aspx`, Dec. 2009. Accessed March 29, 2019.

[23] Lehman, M. M., and Belady, L. A., Eds. *Program Evolution: Processes of Software Change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[24] Letouzey, J.-L. The sqale method for evaluating technical debt. *2012 Third International Workshop on Managing Technical Debt (MTD)* (2012), 31–36.

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

[26] Lim, E., Taksande, N., and Seaman, C. A balancing act: What software practitioners have to say about technical debt. *IEEE Software 29*, 6 (Nov 2012), 22–27.

[27] Marinescu, R. Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development 56*, 5 (Sep. 2012), 9:1–9:13.

[28] Martini, A., Besker, T., and Bosch, J. Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming 163* (03 2018).

[29] McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering SE-2*, 4 (Dec 1976), 308–320.

[30] Morgenthaler, J. D., Gridnev, M., Sauciuc, R., and Bhansali, S. Searching for build debt: Experiences managing technical debt at google. In *2012 Third International Workshop on Managing Technical Debt (MTD)* (June 2012), pp. 1–6.

[31] Norberg, S. Beyond requirements: understanding what the business needs. `https://www.infoworld.com/article/2882015/beyond-requirements-understanding-what-the-business-needs.html`, Feb. 2015. Accessed May 5, 2019.

[32] Nord, R. L., Ozkaya, I., Kruchten, P., and Gonzalez-Rojas, M. In search of a metric for managing architectural technical debt. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture* (Aug 2012), pp. 91–100.

[33] Parnas, D. L. Software aging. In *Proceedings of the 16th International Conference on Software Engineering* (Los Alamitos, CA, USA, 1994), ICSE '94, IEEE Computer Society Press, pp. 279–287.

[34] Power, K. Understanding the impact of technical debt on the capacity and velocity of teams and organizations: Viewing team and organization capacity as a portfolio of real options. In *2013 4th International Workshop on Managing Technical Debt (MTD)* (May 2013), pp. 28–31.

[35] Ramakrishnan, S. Scrum Alliance – Managing Technical Debt. `https://www.scrumalliance.org/community/articles/2013/july/managing-technical-debt`, July 2013. Accessed March 17, 2019.

[36] Ribeiro, L. F., de Freitas Farias, M. A., Mendonça, M. G., and Spínola, R. O. Decision criteria for the payment of technical debt in software projects: A systematic mapping study. In *ICEIS (1)* (2016), pp. 572–579.

[37] Robert C. Martin (Uncle Bob). A Mess is not a Technical Debt. - Clean Coder. `https://sites.google.com/site/unclebobconsultingllc/a-mess-is-not-a-technical-debt`. Accessed March 17, 2019.

[38] Seaman, C., Guo, Y., Izurieta, C., Cai, Y., Zazworka, N., Shull, F., and Vetrò, A. Using technical debt data in decision making: Potential decision approaches. In *Proceedings of the Third International Workshop on Managing Technical Debt* (Piscataway, NJ, USA, 2012), MTD '12, IEEE Press, pp. 45–48.

[39] Tom, E., Aurum, A., and Vidgen, R. An exploration of technical debt. *Journal of Systems and Software 86*, 6 (jun 2013), 1498–1516.

[40] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering PP* (01 2017), 1–1.

[41] Yli-Huumo, J., et al. *The role of technical debt in software development*. PhD thesis, Lappeenranta University of Technology, 2017.

Appendix A

Specific technical debt types

Technical debt type	Characteristics
Architecture debt	Architecture debt — often called structural debt as well — refers to debt resulting from sub-optimal solutions designed at the highest level of a system. For instance, this type of debt can be created when declaring software components (including their roles) and defining relationships between each of them. A more concrete example mentioned by Alves et al. [3] is the lack of modularity. As it was pointed out by Brown et al. [7], due to its structural nature, debt of this kind cannot be paid down by simply changing a few lines of code, since the entire system is influenced by the decisions made at the stage of architectural design.
Build debt	Morgenthaler et al. [30] addressed issues related to build files and the build process in general in their publication. These files essentially define modules of code, source files, dependent libraries and also contain build metadata. Hence, inefficiencies of these artifacts make the build process harder and more time-consuming and represent debt according to Alves et al. [3]. Moreover, Morgenthaler et al. [30] discussed the difficulties of dependency management and also added that build files are manually maintained, which further increases the probability of having such debt.

Code debt	It refers to internal quality issues related to the source code of a system (discussed by Bohnet & Döllner [4]). This type of debt is usually one of the most well-known by developers, since several of them need to address code debt-related issues during their daily work. However, it also means that it is much less visible to stakeholders, which makes it harder to pay down. Just like Alves et al. [3] mentioned, code debt makes the source code harder to read, which consequently results in increased maintenance costs. In other words — as Tom et al. [39] indicated in their article — any part of the code base that needs refactoring can be considered code debt.
Design debt	As per the definition given by Alves et al. [3], design debt refers to those defects of the source code, which go against for instance object-oriented programming design principles. Luckily, these can be identified by analyzing the code base by using simple metrics, such as the coupling of classes or code complexity.
Documentation debt	Tom et al. [39] shared the opinion of Alves: the lack of proper knowledge distribution is also one type of technical debt. In the words of Alves et al. [3], this can mean *"missing, inadequate, or incomplete documentation of any type"*. The role of documentation is key to the successful maintenance and usage of systems, as companies cannot rely on a constant set of employees. People come and go, therefore their knowledge about specific projects should stay within the firm as well in the form of some sort of documentation, even if they decide to go.

Environmental debt	As reported by Tom et al. [39], this type of debt refers to inefficiencies related to the environment of an application. Some of the possible issues involve — but are not limited to — processes, hardware, infrastructure and even supporting applications. As an example, they highlighted the exploitation of harmful security vulnerabilities as a possible result of the postponement of an infrastructure, which can easily accrue technical debt in the form of brand damage.
People debt	Refers to human resources-related issues that can easily hinder efficient software development. In line with what Alves et al. [3] stated, a good example of such problems is having technical expertise concentrated in just a couple of people.
Process debt	Alves et al. [3] defined this type of debt as the inefficiency of processes. As an example, they mentioned processes that become obsolete in time and are poorly maintained. Another good example of process debt is having parts of the process — that could be easily automated — done manually. Consequently, a significant amount of time is wasted on repetitive activities.
Requirement debt	As Alves et al. [3] explained in their paper, requirement debt can be thought of as tradeoffs regarding the set of requirements that developers have to implement. Furthermore, it also has to do with the completeness of implementations. For instance, requirements that are not fully implemented or do not take non-functional requirements (e.g., security, performance) into consideration exemplify requirement debt. In addition, in one of his articles, Ernst [11] defined requirement debt as "the distance between the optimal solution to a requirements problem and the actual solution, with respect to some decision space".

Test automation debt	This type of debt is closely related to testing done as part of continuous integration. Previously implemented and working features should stay functional at all time and this can be validated by running the corresponding tests. However, doing all that manually creates test automation debt. Therefore, tests should be run by continuous integration systems automatically to ensure that integrating new pieces of implementation into the the existing code base does not break other, already existing parts of it.
Test debt	Test debt refers to activities related to the creation of tests. According to the explanation of Tom et al. [39], the most common issue is low test coverage. As a rule of thumb, most developers do not really like to write tests once a feature is implemented. And the incentive is even lower once they no longer remember the details of the implementation. Therefore, having test debt is painful to pay off. Not surprisingly, this is one of the key problems that test-driven development is supposed to solve.
Versioning debt	Alves et al. [2] described this kind of debt as one that is related to the usage of version control system. As an example for the same phenomenon, Greening [16] mentioned the problem of code-forking (where the initially forked code copies are never actually merged back together).