

Arpita Chiddarwar

TOOLS AND TECHNIQUES FOR MANAGING TECHNICAL DEBT

Master of Science Thesis
Information Technology and Communication Sciences
Examiners: Zheyang Zhang
Terhi Kilamo
December 2024

ABSTRACT

Arpita Chiddarwar: Tools and Techniques for managing technical debt
Master of Science Thesis
Tampere University
MSc in Software, web and cloud
December 2024

Technical debt refers to the hidden cost of extra work resulting from opting for a quick or less-than-ideal solution in software development to achieve short-term objectives. While it allows for faster delivery, unmanaged technical debt can lead to significant long-term consequences, such as reduced software quality, increased maintenance costs, and compromised scalability. Managing technical debt effectively is therefore crucial to maintaining sustainable software development practices. Much research is already performed to identify technical debt and apply strategies to pay technical debt off. But it is also important to investigate the current strategies and advanced tools in order to provide software organizations with the information that enables technical debt management in practice. This thesis focuses primarily on two aspects: firstly, the identification of various types of technical debt (TD) using tools, and secondly, the best practices for successfully managing technical debt in a software organization.

The study reveals that tools for identifying technical debt are underutilized, primarily due to a lack of awareness and limited adoption in practice. While numerous tools are available to assist practitioners, selecting the most suitable tool for a specific project can be challenging. To address this, the research also explores strategies to help practitioners identify the right tool for their unique needs. Additionally, the tools are evaluated in detail and categorized to provide a clearer understanding of their strengths and applicability in different scenarios.

This thesis also contributes to a deeper understanding of management practices by performing SLR, offering actionable insights and recommendations for organizations to enhance their TDM processes. By leveraging the right tools and strategies, organizations can effectively address technical debt, improve software quality, and achieve better business outcomes.

Keywords: Case Study, Technical Debt, Technical Debt management, Software Organizations

The originality of this thesis has been checked using the Turnitin OriginalityCheck service.

USE OF AI IN THESIS

Artificial intelligence (AI) has been used in generating this work:

- No
 Yes

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
Chatgpt	4.0
Grammarly	NA

Purpose of the use of AI

ChatGPT was used to improve the paragraph structures and content flow. It was also used for getting the instructions on setting up the tools like Sonarqube and PMD. Grammarly a grammar checking tool was used to ensure grammatical accuracy of the thesis writing

Parts of this work, where AI was used

I used the AI tools for the purpose stated earlier mainly in chapter 3.2 Tools comparison, 4.4 Results and 5 Discussion.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

PREFACE

I would like to express my deepest gratitude to my supervisor, Zheyang Zhang, for her invaluable guidance and unwavering support throughout the journey of completing this thesis. Her insights and expertise have been instrumental in helping me delve deeply into the intricate concept of technical debt. While I had a preliminary understanding of this subject, her mentorship allowed me to explore it more comprehensively as I progressed with my research and writing.

I am also immensely grateful to everyone who contributed to this work in various capacities. Special thanks go to my loved ones—my husband and my family—whose constant encouragement and belief in me provided the strength I needed to pursue this endeavor. Their support has been a cornerstone of my academic journey.

It is my hope that this work contributes to a broader understanding of technical debt, inspiring improved practices in software quality, usability, management, and business outcomes. Let this be a step forward to promote smarter and more efficient strategies to manage technical debt in software development.

Tampere, 31st December 2024

Arpita Chiddarwar

CONTENTS

1.	Introduction	1
1.1	Background	1
1.2	Research Objective	1
1.3	Research Process	2
2.	Related Work	3
2.1	Technical debt overview	3
2.2	Technical debt types	5
2.3	Impacts of TD	7
2.4	Technical Debt Management	9
2.5	TD identification	9
2.6	Agile methodology in technical debt management	11
3.	Advanced tools	13
3.1	Overview of tools in TDM	13
3.1.1	Classification of tools in TDM	13
3.1.2	Evaluation Criteria for selecting a tool	15
3.2	Tools Comparison	16
3.2.1	Objective of the comparison	16
3.2.2	Design of the Comparison	16
3.2.3	Result of comparison	19
3.2.4	Discussion and Reflection	22
4.	Case Studies on Technical Debt Management	24
4.1	Research design and approach	24
4.2	Research process	25
4.3	Case studies	28
4.4	Results	32
5.	Discussion	39
5.1	Technical Debt Identification Using Tools	39
5.2	Effective Management of Technical Debt	40
5.3	Threats to Validity	41
5.3.1	Construct Validity	41
5.3.2	Internal Validity	41
5.3.3	External Validity	42
5.3.4	Conclusion Validity	42
5.4	Future Work	42

6. Conclusion 44
References. 46
Appendix A: Evaluation of tools 52

LIST OF SYMBOLS AND ABBREVIATIONS

Agile	International Organization for Standardization
EMF	Eclipse Modeling Framework
ISO	International Organization for Standardization
JDT	Java Development Tools
LOC	Line of Code
SAT	Static analysis tools
SIG/TUViT	Special Interest Group on Telecommunications and Video/Image Technology
SQALE	Software Quality Assessment Based on Lifecycle Expectations
TAU	Tampere University
TD	Technical Debt
TDD	Technical Debt Diversity
TDM	Technical Debt Management
TUNI	Tampere Universities
URL	Uniform Resource Locator

1. INTRODUCTION

1.1 Background

The term technical debt(TD) was first coined by Cunningham[1] for the accumulated additional rework to be done on software tasks. These software tasks accumulate because of taking shortcuts in developing software. In an organization, a software developer takes a quicker approach to deliver the project swiftly, resulting in short-term output but long-term impact[2]. According to [2] technical debt often accumulates gradually as developers make trade-offs between short-term gains and long-term maintainability. Over time this affects the quality of software in terms of adherence to the organization's coding standards, documentation, and how easy code is to understand[3]. In an organization, software maintenance team managers or leaders estimate costs, and resources and trace risks that hinder the quality[4]. In large systems, delayed tasks are often overlooked, or their impact is misjudged [4]. This frequently results in unexpected delays in implementing required changes and a decline in overall quality. The technical debt term comes into the picture when these delayed tasks often provide short-term gain(debt) but later the "debt" has to be paid off which means this software task has to be completed along with "interest", here interest is additional efforts and decreased productivity[5].

According to [2] in large-scale organizations the impact of paying back the technical debt can be costly. Technical debt poses risks to the organization, such as increased maintenance costs, reduced agility, and decreased software quality. By detecting technical debt early and assessing its potential impact, organizations can better manage these risks and make informed decisions about resource allocation and project planning. A tool for detecting technical debt can help estimate the true cost of maintaining and refactoring the codebase, enabling organizations to allocate resources more effectively[6].

1.2 Research Objective

The primary objective of this thesis is to investigate strategies and methodologies for detecting and managing technical debt in its early stages within large organizations. The study aims to explore various tools, techniques, and best practices employed for the early detection of technical debt, with a focus on their effectiveness and applicability in real-world scenarios. In a large organization, it is important for the regular detection and man-

agement of such tasks because unaddressed technical debt leads to higher maintenance costs in the long run. As the codebase becomes more complex and difficult to maintain, developers spend more time fixing bugs and which in turn decreases productivity. This research tackles the following research questions. :

RQ1: What are the tools to identify the technical debt?

The first research question evaluates a range of tools commonly used for detecting technical debt in software projects. The question assesses their capabilities and limitations for detecting different types of technical debt.

RQ2: How can an organization manage technical debt successfully?

The second research question aims to provide insights and guidelines for organizations through a systematic literature review. The findings aim to enhance any software organization's ability to detect and manage technical debt effectively throughout the software development lifecycle.

1.3 Research Process

In addressing the research questions, a case study approach is used. The research began with a comprehensive literature review in Chapter 2, which involved studying existing academic literature to gather insights into technical debt, types of debt, causes and impact. This literature review identified existing methodologies and best practices regarding technical debt detection.

The range of tools and techniques that can effectively detect technical debt are discussed in Chapter 3 which answers RQ1. In chapter 3 we also discussed evaluation criteria for selecting the right tool for the project's needs. We also compared two tools as per the derived evaluation criteria. Research design and approach, in-depth research process, case study analysis is presented in Chapter 4 to answer RQ2.

Findings from the evaluation of tools and SLR(Systematic Literature Review) is summarized in chapter 5 to answer RQ1 and RQ2. We discuss the limitations and future work of the thesis in chapter 5 as well. The conclusion in chapter 6 summarizes the research.

2. RELATED WORK

2.1 Technical debt overview

The concept of technical debt was first introduced as a metaphor by Ward Cunningham in 1992 [1], describing the eventual consequences of shortcuts or suboptimal solutions in software development. Over time, these shortcuts accumulate and create a "debt" that must be repaid through additional work. This literature review explores the evolution of the concept, its impact on software development, types of technical debt, and emerging research trends. Cunningham's metaphor of technical debt has evolved significantly since its inception. Initially, technical debt was primarily understood in the context of code-level issues. Early discussions focused on "code debt," which includes poor coding practices that lead to high maintenance costs and increased complexity. Over time, the scope of technical debt expanded to include broader software engineering aspects, such as design, architecture, testing, and documentation[7]. Martin Fowler and other thought leaders in software development played a significant role in broadening the concept. Fowler [8] emphasized the importance of refactoring to manage technical debt, introducing the idea that continuous improvement of code quality is necessary to prevent debt from becoming unmanageable.

Debt can encompass any element of the software identified as suboptimal but not immediately addressable. Examples include missing or outdated documentation, unperformed planned tests, excessively complex code needing refactoring, and unresolved defects. These incomplete artifacts often cause unforeseen delays in implementing required changes and pose challenges in achieving the project's quality standards [9][10].

Technical debt often arises in software projects when there is a trade-off between ensuring system quality and quickly delivering the software with limited resources.[2]. These instances of TD, often referred to as "items," may necessitate debt repayment with penalty later in the project. Translating this metaphor into a model for analysis, following variables were identified:

- Principal of the debt: This is the cost of resolving the debt, which reflects the effort needed to complete the task that was previously postponed [2].

- Interest amount: This refers to the potential penalty incurred due to delayed task completion, leading to increased effort and reduced productivity in the future. It also accounts for the higher cost of addressing the debt later rather than sooner [11].
- Interest probability: This represents the likelihood that technical debt will have a negative impact on future project activities. For instance, if the artifact associated with the debt is likely to require maintenance, the chances of the interest negatively affecting the project increase.

Despite similarities between the terms and concepts, technical debt is not identical to financial debt. The key difference is that the interest associated with technical debt may not always need to be repaid [9]. To maintain short-term productivity while tracking project progress to prevent incurred debt from hindering development, technical debt (TD) management techniques have been developed [11]. These techniques focus on identifying and monitoring TD items to make them explicit and ensure they are addressed at the appropriate time.

In the blog post "The Future of Managing Technical Debt" Robert Nord discussed the conceptual model of technical debt [12]. The conceptual model serves the purpose of differentiating the TD from other software anomalies. The conceptual model can be used to understand core concepts of TD that are important to make progress on the TD quantification.

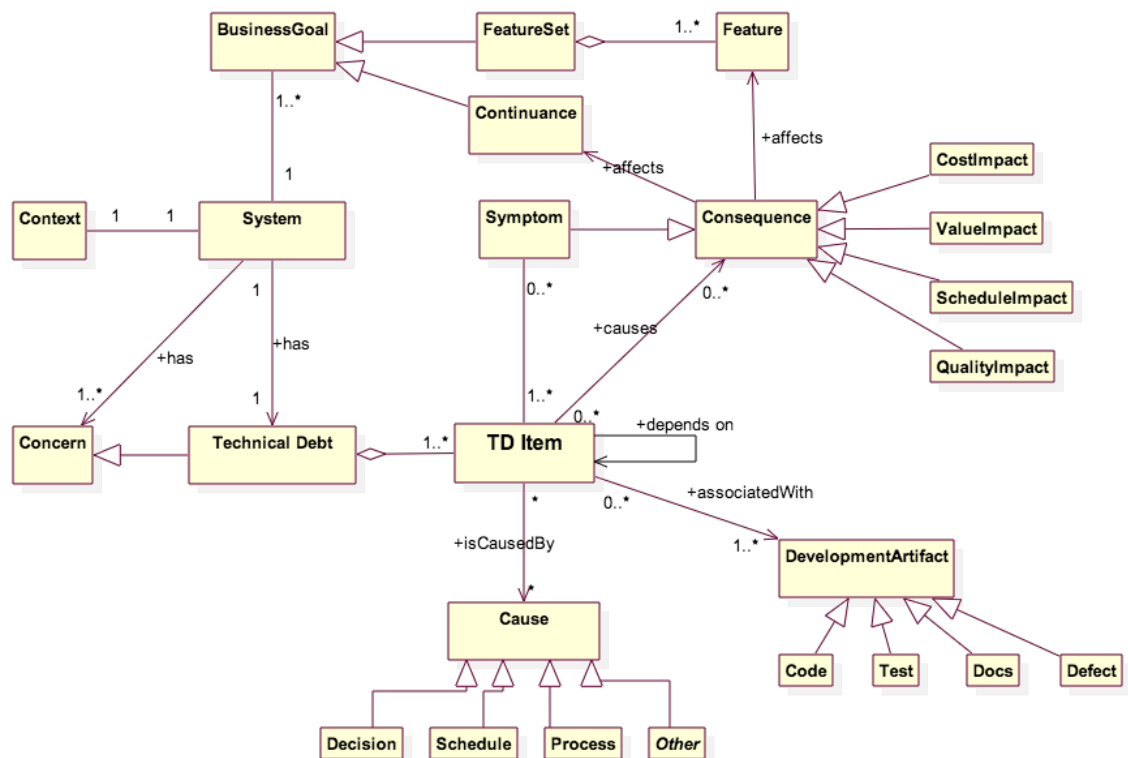


Figure 2.1. Conceptual model of technical debt to aid debt quantification [12]

In this conceptual model, the highlighted word Technical debt is associated with the complex software System. This technical debt is composed of Technical Debt items(TD Item). The TD item has causes and consequences. The causes of TD Item that produces the Technical Debt in the system are poor decision-making by management, schedules, insufficient development process, unknowledgeable persons about the technical feature, and so on.

There are many negative consequences of technical debt: the first being the impact on productivity which in turn affects the cost, the schedules of the deliverables are affected, and through schedule delays, the code quality is degraded.

The TD item is associated with several tangible artifacts in the software development process such as code and architectural designs, test suites build suites, and documentation. According to Robert Nord[12] these tangible artifacts can be managed and monitored to detect the defects associated with the system.

The conceptual model is the basis for understanding the classification of TD, what artifacts to look at and what kind of techniques are needed to analyze the TD. In the further sections, I have discussed the classifications of TD derived from the conceptual model.

2.2 Technical debt types

Technical debt can be divided into two categories [13] such as unintentional technical debt and intentional debt. The term "unintentional technical debt" refers to modifications performed in a hurry without considering refactoring the code. Inadequate planning due to unskilled employees or environmental changes is a prevalent cause of unintentional TD, which happens spontaneously and non-strategically [14]. It may also be the consequence of bad design choices brought on by ignorance or disregard for development guidelines. In the context of testing, unintended technical debt happens when testing is not conducted properly. The other category presented in [13] is Intentional TD which occurs when professionals intentionally and strategically choose to take shortcuts, find other solutions, or leave jobs unfinished in order to gain short-term advantages. Technical debt that is purposeful is chosen consciously. It needs to be planned for refactoring and documented. For instance: A regional sales manager has a report that has to be generated by a certain date, but the platform isn't able to do it. To make the deadline, the manager starts using an open-source report writer. It is known as technical debt [15]. However, it is a deliberate technical debt if the development team invests money to replace the master reporting system and eliminate the open-source report writer. According to [11] this type of category of TD is helpful to find the causes of TD, which further can lead to various identification approaches.

Furthermore, Li et al.[16] carried out a systematic mapping study to better understand

the concept of technical debt (TD) and provide an overview of the existing research on TD management. From the 96 selected studies, Li et al.[16] developed a classification scheme that identifies 10 distinct types of TD across various levels of software lifecycle. This classification highlights the potential sources and types of technical debt, each of which may require distinct methods for detection and strategies for management. First, Requirements TD represents the gap between ideal requirements specifications and the actual implementation of the system, considering domain constraints. Architectural TD arises from architectural decisions that compromise certain internal quality factors, like maintainability. Design TD involves shortcuts taken during the detailed design phase. Code TD is characterized by poorly written code that disregards best practices, including issues like code duplication and excessive complexity. Test TD encompasses shortcuts in testing processes, such as a lack of essential tests, including unit, integration, and acceptance tests. Build TD relates to flaws in a software system, its build system, or the build process itself, leading to unnecessary complexity. Documentation TD pertains to inadequate, incomplete, or outdated documentation throughout the software development lifecycle, such as obsolete architectural documents or insufficient code comments. Infrastructure TD involves sub-optimal configurations of processes, technologies, and tools that hinder a team's capacity to deliver quality products. Versioning TD addresses issues with source code versioning, such as unnecessary forks in the codebase. Finally, Defect TD refers to the presence of defects, bugs, or failures within software systems.

However, in this research, we focus on the four main TD types. These TD types or dimensions of TD are the most commonly analyzed in the research papers. According to Alves et al. [3] the papers from 2006 to 2014 mostly concentrate on architecture, design, and documentation debt, along with some papers on code and test debt. The figure 2.2 shows the number of research papers that discuss each TD type.

TD Type	2006	2010	2011	2012	2013	2014	Total
Design	1	5	8	11	9	8	42
Architecture	0	2	3	11	5	9	30
Documentation	0	2	4	6	4	12	28
Test	0	2	2	8	6	6	24
(Type not specified)	0	1	1	5	6	10	23
Technical debt							
Code	0	3	1	9	5	3	21
Defect	0	1	5	3	3	5	17
Requirement	0	0	0	2	0	2	4
Infrastructure	0	1	0	1	1	0	3
People	0	0	0	1	0	2	3
Test automation	0	0	0	0	2	1	3
Process	0	0	0	0	2	1	3
Build	0	0	0	1	0	1	2
Service	0	0	0	0	2	0	2
Usability	0	0	0	0	1	1	2
Versioning	0	0	0	0	1	0	1

Figure 2.2. Papers by type of TD over the years.[3]

Ampatzoglou et al.[17] carried out a multiple case study in the embedded systems sector

to examine the anticipated lifespan of components impacted by technical debt (TD) and identify the most common forms of TD. The study involved seven companies from five different countries. The findings revealed that the most prevalent types of TD identified were related to testing, architecture, and code. Prathibhan[18] in his research examining the tools for managing different dimensions of Technical Debt has discussed four main dimensions of technical debt: code debt, design debt, test debt and architecture debt. [18] provides the categorization of the technical debt items associated with their technical debt dimension. The TD types code, design, test and architecture and the corresponding TD items are shown in the Table 2.1

TD type	TD items
Code debt	Coding guideline violations, Code smells, Inconsistent style
Design debt	Design rule violations, Design smells, Violation of design constraints
Test debt	Lack of tests, Inadequate test coverage, Improper test design
Architecture debt	Architecture rule violations, Modularity violations, Architecture smells

Table 2.1. *Categorization of TD items[18]*

Code debt is a collective term that refers to poor quality or suboptimal code and whose structure is hard to understand, modify, or even develop further. This might be missed or skipped during coding sessions, failure to follow coding standards, or rarely refactoring or code deprecation or using outdated libraries or frameworks [19]. Alves et al. [3] discuss that the design debt includes mistakes in the system architecture or design that render the system more difficult to alter, integrate, or re-configure. Lack of careful planning at the start, lack of vision as to what the software should embody, changes in the requirement, or changes in technology are the main causes. Architecture Debt means issues with the overall system design, such as the use of outdated or incompatible technologies, lack of scalability, or poor modularization [3].

Test debt is caused by poor test coverage, inadequate tests or no tests at all meaning a higher probability of defects and lower assurance for customers on structures in question. Some common instances include; Constant aggregation of tests in the name of meeting deadlines, low test coverage, and reliance on manual testing instead of automated [20].

2.3 Impacts of TD

As technological debt accumulates, the team's capacity to improve the system properly becomes increasingly challenging[5]. This can be the result of bad code that is hard to modify today or bad feature documentation done by previous tech teams during the development cycle. Inevitably, such technical debt can have significant impacts on firm performance[21]. Fig. 2.3 summarizes the impacts of TD on code, organization's product,

team and business.

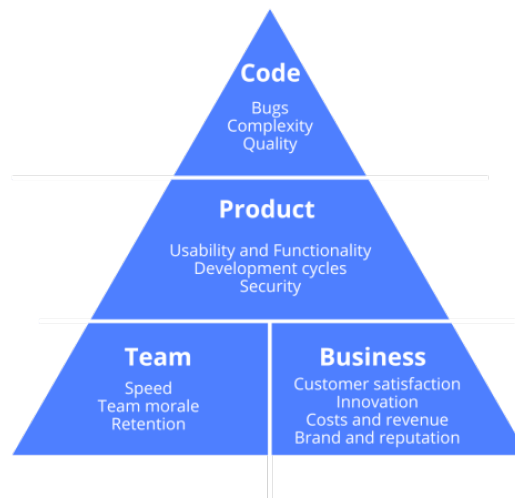


Figure 2.3. Impacts of TD

Research suggests that technical debt can have both positive and negative implications for firm performance[21]. On the one hand, incurring technical debt can allow firms to expedite system deployment and implement unique functionalities that may enhance performance[21]. On the other hand, technical debt can lead to system disruptions that impair firm-level performance[21]. Additionally, the extent to which a firm accumulates technical debt may depend on the industry and competitive dynamics it faces. Companies in software industries with faster "clockspeed" and greater competitive pressures are more likely to accumulate technical debt [21]. Systems with larger technical debt are more error-prone, resulting in a greater number of defects. Furthermore, technical debt can impede a firm's ability to react quickly to new demands and opportunities in the market[21][5].

Legacy code that was implemented in a suboptimal manner may require significant refactoring efforts before additional development can be effectively undertaken[22]. Technical debt accrued through such suboptimal design decisions can accumulate over time, making the codebase increasingly complex and challenging to maintain[23]. Addressing this technical debt through strategic refactoring is often necessary to enable further development efforts on the system. The team's productivity may decrease over time as more technical debt accumulates, necessitating a deliberate approach to managing this debt[5]. Often this technical debt and security vulnerabilities are related, making security issues difficult to address[24]

2.4 Technical Debt Management

As software systems grow in complexity and scale, the need to effectively manage TD becomes increasingly critical for successful software delivery[25]. Maintaining low levels of technical debt (TD) in critical areas of a software system, particularly in design hotspots—modules with a high likelihood of accruing interest—is crucial for ensuring system quality and sustainability[26]. The practice of controlling and managing TD throughout the lifecycle of a software system is known as technical debt management (TDM) [26]. Preventing the accumulation of TD using tools as the software evolves is a key objective in this research.

Effective TDM, as outlined by Li et al. [27], involves a set of activities. The first step is identifying the software artifacts, such as classes, that are affected by TD. Once identified, the second step is to quantify the TD, often translating it into monetary terms to measure the principal, interest, and the likelihood of accruing interest. Prioritizing these TD items based on the urgency of their resolution is the third step and it is essential to manage them effectively. The fourth step is the prevention of further TD accumulation, which requires proactive strategies to avoid the introduction of new debt. Additionally, continuous monitoring of TD is necessary to track its status throughout the project's evolution. Repayment, often achieved through refactoring, addresses the existing TD by resolving the issues associated with it. Documenting and visualizing all relevant TDM data ensures that the process remains transparent and traceable. Finally, effective communication of TD-related information is vital to keep all stakeholders informed about the status and management efforts, promoting a shared understanding and collaborative approach to addressing technical debt.

Similarly, as per the researchers, TDM encompasses a range of activities that are performed to identify, measure, prioritize, and address technical debt within software systems[4]. These activities typically involve code and architectural analysis to detect areas of technical debt, as well as the development of strategies and plans to gradually reduce or manage this debt over time[4]. These activities often involve tasks like code and architectural analysis, which can be highly time-intensive when performed manually. Therefore, this research has focused on studying automated tools to simplify and speed up the tasks involved in TDM. These tasks usually involve reviewing the code and software design to find areas with technical debt.

2.5 TD identification

There can be multiple reasons why TD occurs in the system but despite that it is important to manage the occurred TD so that it does not hinder the quality and growth of the project[3]. Identification is the first step to avoid the late discovery of TD and to avoid the

correction cost[27]. According to [2] TD identification is potentially significant because this activity helps to know what type of TD exists and where it is located. Many studies have presented strategies and tools for identifying common TD indicators and evaluators to assess and measure TD. Alves et al. [3] conducted research to identify the common TD indicators and strategies to manage the TD. Alves et al.[3] found that code smells were the most common indicator. Fowler first used the term "code smells," which describes design decisions made in object-oriented systems that depart from recognised guidelines such as information encapsulation and concealing. Three categories of code smell exist: disharmonious classification, identification, and collaboration. To find code smells, automatic detection techniques have been created. The accuracy and recall of these automated methods were assessed by Zazworka et al. (2014) [43]. The study by Olbrich et al. (2010) [44] investigated the connection between software component defect and change proneness and code smells, including god classes. The researchers discovered that in industrial settings, automatic classifiers for god classes show good recall and precision. Furthermore, compared to the non-smelly counterparts, deity classes had a considerably higher likelihood of being affected by flaws and changes—up to 13 times and seven times more, respectively.

The second common indicators found in a system are Code Comments and Defect/Bug[28] while Alves[3] suggests the most common indicators are Violations of modularity and Automatic static analysis(ASA) issues. A fundamental principle of software engineering is modularity, which states that a system should be divided into smaller, independent components that can be developed, tested, and maintained separately[29]. However, over time, large-scale business applications often deviate from this intended architecture and deteriorate into unmanageable monoliths. As the system grows in complexity, it becomes increasingly difficult to make changes or additions without introducing bugs or unintended side effects[29]. There are several reasons why modularity violations occur. The concurrent development of modules that should, in theory, evolve independently is one typical explanation. This situation points to a systemic concealed reliance or undesired side effect. A lot of things can lead to this kind of simultaneous evolution, such as hurriedly writing code without thinking through its long-term effects, remaining experimental code that ought to have been eliminated. [3] suggested Automatic static analysis issues as the other common TD indicator. Automatic static analysis issues refer to problems or warnings generated by static analysis tools, which analyze a software codebase without executing it. Static analysis is useful for identifying technical debt, as it highlights problematic areas in the code that may require refactoring [3]. Automatic Static analysis (ASA) tools, such as FindBugs, automatically detect "issues," or deviations from standard programming techniques, which can cause errors or deteriorate software quality attributes like efficiency and maintainability [30]. We discussed about ASA tools in detail in next sections.

Zazworka et al. in their study which compares the technical debt identification approaches [29], suggested additional indicators, such as design patterns and grime buildup. Many factors contribute to the popularity of design patterns, such as the assertion that they are more flexible and easier to maintain, that there are fewer errors and defects [31], and that the result in better architectural designs. As systems, users, and operating environments change, software designs deteriorate; this degradation might include design patterns. Classes that take part in the realisation of design patterns amass dirt, or code unrelated to patterns. Understandings of design patterns can also degrade when modifications compromise the structural or functional integrity. Rot and filth are examples of TD. Izurieta and Bieman (2007) [32] conducted a pilot research on the impacts of decay on a tiny portion of the open-source system JRefractory and offered the idea of design pattern grime. Studying a limited number of pattern realisations, Izurieta and Bieman (2007) discovered that design pattern filth caused coupling to rise and namespace management to become more complex. However, the researchers were unable to identify any alterations that would "break" the pattern (design pattern rot). Izurieta and Bieman (2007) [32] also looked at how design pattern grime affected JRefractory's testability. After looking at a few patterns, Izurieta and Bieman (2007) discovered that testability can be impacted by at least two different mechanisms: First, there are design anti-patterns emerging, and second, there are more interactions (dependencies, realisations, and associations) which raise the need for tests. The study also discovered that coupling increases account for most of the dirt accumulation. Grime buildup refers to the gradual accumulation of poor design or implementation practices in software code over time, specifically within the context of object-oriented design patterns. It occurs when code meant to follow certain design patterns becomes cluttered with small, incremental changes that deviate from the original design intent. These changes may not initially seem harmful but, over time, they degrade the code's structure, making it harder to maintain and increasing technical debt. Grime buildup is a form of design decay, where the code still adheres to a pattern but becomes less clean and more difficult to work with due to unnecessary complexity or rule violations. This concept highlights the importance of not only following design patterns but also maintaining their integrity through regular refactoring.

2.6 Agile methodology in technical debt management

Agile software development makes extensive use of the technical debt concept to mitigate the effects of working in extremely short cycles [33]. In this context, instances of technical debt involve executing only those aspects of a norm that are required for present use, rather than fully complying, or adding capabilities to an existing too big module instead of splitting it up into smaller, simpler modules. The effects on subsequent work are cumulative in both of these situations. The situation deteriorates with each addition to the excessively large module in the first instance, making it harder to comprehend, alter, and

test. In the second scenario, inadequate adherence to standards may lead to integration challenges and promote shoddy solutions, thereby diminishing the product's overall quality [33]. Agile approaches, while potentially contributing to technological debt, provide effective methods for managing and reducing it. Agile promotes frequent evaluation on ways to increase efficiency, which can be used to manage technology debt by adjusting practice as necessary. For example, frequent, small releases are highly valued in agile methodologies. This approach allows teams to identify issues and take action before they become more serious. They can also refactor as they go, which gradually improves the codebase and reduces technical debt. Moreover, Agile's focus on collaboration encourages team members to share information, which reduces the likelihood of knowledge silos, which can lead to technological debt. Maintaining clean, debt-free code is made easier when everyone is familiar with the source [34].

3. ADVANCED TOOLS

3.1 Overview of tools in TDM

In software engineering, many tools are already available for technical debt management, but the question is as to what particular tool should be used in a particular software project[35]. The current research [36] highlights that the majority of tools for managing technical debt (TD) primarily focus on three key types: source code issues, which are addressed by 60% of the tools examined (30 out of 50), architectural issues, targeted by 40% of the tools (20 out of 50), and design issues, covered by 28% of the tools (14 out of 50). These findings align with those of earlier studies conducted by Rios et al. (2018) [37] and Li et al. (2015) [27], which similarly identified source code, architecture, and design as the predominant areas of interest among researchers developing tools for technical debt management.

This consistent focus on source code, architecture, and design in both current and previous research suggests that these TD types remain central to the ongoing challenges faced by developers and organizations in managing technical debt. Given their critical impact on software maintainability, performance, and scalability, tools addressing these types of debt play a significant role in improving the long-term health of software systems. As a result, this study will emphasize tools that effectively address these areas of technical debt, as they represent the most pressing concerns within the field and continue to attract the attention of both researchers and practitioners.

3.1.1 Classification of tools in TDM

TDM tools typically belong to distinct categories, where each targets the specific stages of organizational needs[36]. However, this thesis primarily focuses on four types of technical debt: Code TD, Design TD, Architecture TD, and Test TD. To address these, I identified three categories of TD identification tools based on evaluation studies conducted by [27], [36], [38], and [39]. These categories include Static Analysis Tools, Architecture Analysis Tools, and Test Management Tools, which are specifically designed to identify and address the aforementioned technical debt types.

Static code analysis is a method of examining computer programs without executing the

software [40]. This technique is often employed to identify potential problems, defects, security vulnerabilities, or deviations from coding standards, all of which can contribute to the accumulation of code TD within the software system[38]. By analyzing the source code statically, developers can detect issues early in the development lifecycle, allowing them to address these problems before they manifest into larger technical debt concerns that can impact the long-term maintainability and sustainability of the software project [41][3].

According to study done by Stefanović et al. in [41] and [40] the most common static analysis tools studied in the recent pieces of literature are given in the Figure 3.1. These tools are most often analyzed and they are presented in 3 or more times in research. The detailed evaluation of these tools is presented in the Appendix A Table A.1 based on the official documents available on the tool's website and comments from the community who have used the tools.

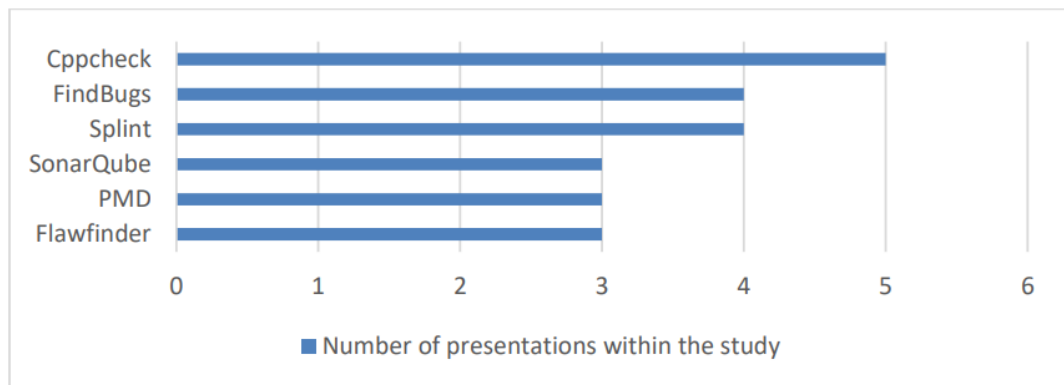


Figure 3.1. The most commonly featured static code analysis tools in the primary studies[41]

Architectural analysis tools focus on evaluating the design and framework of software systems[42]. They help identify architectural problems and potential design flaws that could lead to technical debt[36]. According to [43], many tools are available that identify these two particular TD types: design TD and architecture TD. These tools detect TD indicators such as Architecture smells, Grime, Violations of good architectural and/or design patterns, and Code smells. [36] has provided a list of tools that help in the identification of design and architectural TD. Architectural and design analysis tools such as Arcan, Designite, Lattix, Sonargraph, JSPIRIT are evaluated in detail in Appendix A: Table A.2

Test management tools help identify test debt. Test debt refers to the problems or issues that may affect the quality of testing activities[38]. These tools include frameworks for unit testing, systems for examining user interfaces, and various other testing packages[18]. The primary goal is to ensure thorough testing across all aspects of the software, reducing

the risk of introducing technical issues when code is modified or updated. The evaluation of two test management tools is present in Appendix A : TableA.3

3.1.2 Evaluation Criteria for selecting a tool

Selecting the right tools for your software development is a critical step in ensuring the successful execution of a project[38][35]. The existing tools employ various techniques to identify problematic files, such as detecting "hot-spots" or using co-change information to define anti-patterns[44]. However, the lack of uniformity in these approaches suggests that the tools may not consistently identify the same set of problematic files [35]. This prompts the question of which tools and metrics are most effective in identifying genuine technical debt that demands attention [44]. Therefore, the evaluation criteria outlined below are crucial when selecting an appropriate tool for detecting technical debt.

1. Technology and languages

Not all tools support the programming languages a company requires. Therefore, organizations must evaluate which programming languages a tool supports to ensure alignment with their needs.

2. Features

One important evaluation criteria is the functionalities that a tool offers. The functionality and features can be evaluated by reading the documents available on official website and by trying out if demo licenses are available.[35]

3. Integration capabilities

Another key evaluation factor for a technical debt management tool is its ability to integrate seamlessly with existing systems and workflows. The ability to seamlessly integrate with existing tools and workflows is crucial for adoption and effectiveness. If a tool requires significant effort to set up and integrate, it is less likely to be used consistently by the development team. Tools that can integrate with common development tools such as version control systems, issue trackers, and CI/CD pipelines will be more valuable as they can provide a more holistic view of technical debt within the overall software development lifecycle.

4. Costing

The tool's pricing model should align with the organization's budget and the value it provides in terms of identifying and managing technical debt.

5. Depth and breadth of data

While choosing a tool checking tool's capability to deliver insights on code smells, vulnerabilities, test coverage, and an overall technical debt score. [44]

3.2 Tools Comparison

3.2.1 Objective of the comparison

Technical debt identification involves tools that employ varying terminologies, metrics, and methods, creating significant diversity in how technical debt is defined and assessed [35] and there is a wide range of tools available to select from. As the number of tools continues to grow, practitioners face increasing challenges in selecting the right tool for their requirements [40].

This chapter focuses on comparing the features of two tools within the static code analysis category. By narrowing the scope to two tools, we aim to provide a detailed yet manageable examination of their capabilities. The comparison is based on the evaluation criteria outlined in Section 3.1.2, to offer a clear and practical framework for practitioners to evaluate and select a tool that aligns with their needs. This targeted approach helps illustrate how a methodical assessment can aid in navigating the diverse landscape of technical debt identification tools.

3.2.2 Design of the Comparison

A Tools selection

For this study, I selected two widely used static code analysis tools, namely SonarQube and PMD. Both tools are effective in detecting code smells. I use code smell detection to identify technical debt in a system because code smells are indicators of poor design or implementation practices that can compromise code quality and lead to long-term maintenance challenges. SonarQube and PMD provide valuable insights into code quality and both support Java language but differ significantly in their features, approaches, and capabilities. Hence despite both the tools being from the same category and the purpose of the tools being similar, these tools represent two distinct yet overlapping approaches to static code analysis, making them ideal for a thorough and insightful comparison.

SonarQube[45] is a popular open-source static code analysis tool designed to identify and resolve code quality issues. It can be accessed as a hosted service via the SonarCloud.io platform or installed and operated on a private server. SonarQube assesses various metrics, such as code complexity and the number of lines of code, while ensuring compliance with predefined coding standards for widely used programming languages. Violations of these rules are flagged as "issues," with the tool estimating the effort required to address them, referred to as remediation effort.

SonarQube categorizes its rules into three key areas: reliability, maintainability, and

security. Reliability rules, commonly referred to as "Bugs," identify errors in the code that are likely to result in functional bugs. Maintainability issues, or "Code Smells," are problems that reduce the readability and modifiability of the code.

PMD[46] is a static code analysis tool primarily tailored for Java and Apex, functioning through the command line using its binary distributions. It assesses code quality based on a predefined set of rules, addressing key issues such as unused variables, empty catch blocks, and unnecessary object creation. For Java projects, PMD provides 33 predefined rule sets, which can be customized to suit specific requirements.

The rules are categorized into eight groups: best practices, code style, design, documentation, error-prone, multi-threading, performance, and security. Each rule violation is assigned a priority level ranging from 1 (most critical) to 5 (least severe).

B Tools setup

Both tools support Java, making them suitable for comparison in this study. To evaluate their effectiveness, we analyzed an existing project, OpenMRS-core, available on GitHub [47]. OpenMRS-core is a Java-based project that serves as a patient-centered medical record system, designed to provide healthcare providers with a free, customizable electronic medical record (EMR) solution. This publicly accessible project was chosen for its practical relevance and its suitability for static code analysis.

To set up and configure SonarQube, I installed Docker on the Ubuntu machine to enable containerized deployment. I then pulled the SonarQube v9.9.7 community LTS image using Docker and started the SonarQube instance. Next, I cloned the OpenMRS-core repository from GitHub [47] onto the Ubuntu machine and connected it to the running SonarQube instance for analysis. Finally, I configured a custom ruleset on the SonarQube dashboard to tailor the analysis to the study's requirements. Rulesets refer to a collection of coding rules that the tool applies to analyze source code for issues like code smells. I have created the custom ruleset on the SonarQube dashboard for detecting code smells such as Cyclomatic complexity, cognitive complexity, etc. I kept the threshold values for SonarQube and PMD similar. For example, the threshold for cyclomatic complexity is set to 10, the threshold for cognitive complexity is set to 15, maximum number of allowed methods in a class is set to 20. The image shows the custom ruleset for SonarQube.

To set up PMD, I began by downloading the official PMD package from its website. I then configured it as an environment variable on my Ubuntu machine to ensure system-wide accessibility. After completing the setup, I created a custom ruleset

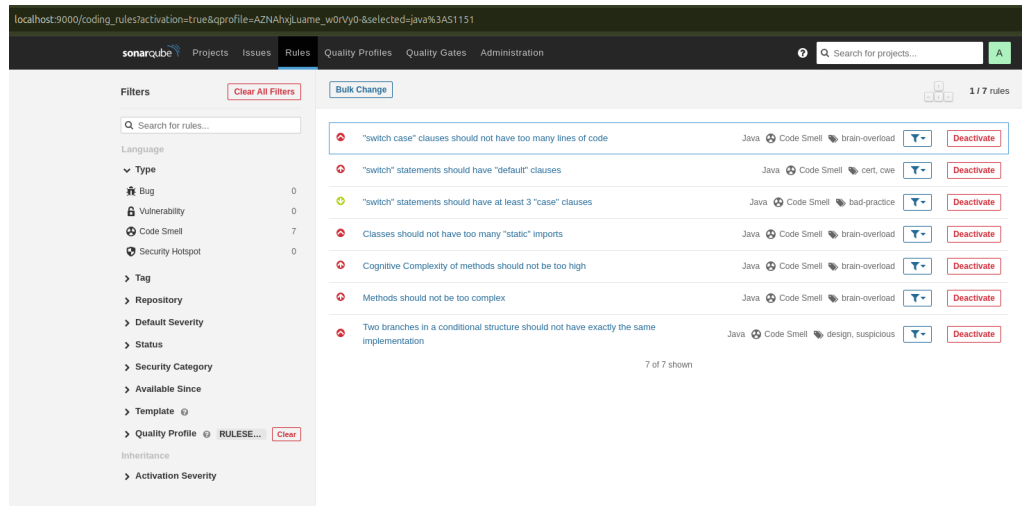


Figure 3.2. SonarQube Ruleset

in a XML file and placed the ruleset file in the PMD folder. The method to define ruleset is quite different in PMD but I created a similar ruleset to maintain uniformity in comparison. Similar to the Sonarqube the threshold values for cyclomatic complexity, and cognitive complexity is set 10 and 15 respectively. The image 3.3 shows ruleset.xml which is created for PMD. With PMD properly configured, I could seamlessly open my cloned repository in Visual Studio Code and execute PMD directly from the VS Code terminal.

C Criteria used for comparison

For comparing two tools I have considered the evaluation criteria written in section 3.1.2. When comparing SonarQube and PMD, selecting factors like programming language support, integration capabilities, features, cost, and the depth and breadth of data will give an overall view of tools capabilities. Different projects use a variety of programming languages, and a tool's ability to support multiple languages directly impacts its relevance. Integration capabilities are vital for assessing how well the tools fit into modern software development workflows. Another important factor while comparing these tools are the set of features each tool offers. Lastly, the comparison is made based on the code smell detection capabilities. Code smells are indicators of potential problems in the source code of a software system. The term "code smells" is used as a metaphor to signify quality issues in source code[48]. A high number of code smells within a software system is linked to significant technical debt, which impedes the system's development and progress[49]. Both the tools are static code analysis tools and specialize in detecting code quality issues such as code smells using customizable rulesets and hence this will be a good comparison factor to investigate tool's capability. The comparison in this study focuses on evaluating the ability of each tool to detect rule violations. Rulesets, which are collections of criteria designed to analyze source code for specific issues, form the

```

<ruleset name="Code Smells Categories"
  <rule ref="category/java/design.xml/GodClass"/>
  <rule ref="category/java/design.xml/TooManyFields"/>
  <rule ref="category/java/design.xml/TooManyMethods">
    <properties>
      <property name="maxmethods" value="20"/>
    </properties>
  </rule>
  <rule ref="category/java/design.xml/ExcessivePublicCount"/>

  <!-- God Method Detection Rules -->
  <rule ref="category/java/design.xml/CyclomaticComplexity">
    <properties>
      <property name="methodReportLevel" value="10"/>
    </properties>
  </rule>
  <rule ref="category/java/design.xml/NcssCount">
    <properties>
      <property name="methodReportLevel" value="50"/>
    </properties>
  </rule>
  <rule ref="category/java/design.xml/CognitiveComplexity">
    <properties>
      <property name="reportLevel" value="15"/>
    </properties>
  </rule>

  <!-- Feature Envy Detection Rules -->
  <rule ref="category/java/design.xml/LawOfDemeter"/>
  <rule ref="category/java/bestpractices.xml/SwitchStmtsShouldHaveDefault"/>
  <rule ref="category/java/design.xml/CouplingBetweenObjects"/>

```

Figure 3.3. PMD Ruleset

basis of this evaluation. For consistency, I used customizable rulesets configured to be as similar as possible in both tools, alongside their default rulesets. The inclusion of similar rulesets provides a reliable basis for assessing the precision of each tool, making it an effective factor for comparison. This approach ensures that the analysis highlights not only the tools' default capabilities but also their adaptability to user-defined configurations, offering a more comprehensive evaluation.

3.2.3 Result of comparison

In this section, I discuss the source code quality issues identified by the tools. I begin by summarizing the evaluation criteria of the tools. To do this, I referred to the tools' documentation to gather information about their key features. Specifically, I compiled a list of the programming languages each tool supports and the kinds of issues they address. For programming language support, we relied directly on the details provided in the documentation. To conduct a detailed comparison of SonarQube and PMD, I independently applied both tools to analyze the OpenMRS Java project[47], a patient-centered medical record platform. Using the evaluation factors outlined in Section 3.1.2, I systematically examined the results from each tool and created a comparison table to highlight their dif-

ferences and strengths. The evaluation factors included programming language support, integration capabilities, features, cost, and the depth and breadth of insights provided.

Evaluation Factor	SonarQube	PMD
Programming Language Support	Supports 27+ languages via plugins	Primarily Java and Apex, with support for 16 other languages
Integration Capabilities	Strong integration with CI/CD pipelines and DevOps tools	Focused on IDE integration and command-line operations
Features	Advanced dashboards, technical debt scoring, and vulnerability analysis, duplicate code detection, code coverage, Sonarlint IDE integration	Contains 400+ built in rules and extended with custom ruleset, copy-paste-detector, built-in checks
Cost	Offers both free/community editions and paid enterprise versions	Completely free and open-source
Depth and Breadth of Data	Provides detailed insights on code smells, vulnerabilities, test coverage, and maintainability	Focuses primarily on detecting specific code smells and rule violations

Table 3.2. Comparison of SonarQube and PMD based on evaluation factors.

After outlining the tools' features, I analyzed their code smell detection capabilities by calculating the number of issues produced by rule violations within the reference dataset. Here, the reference data is a Java project namely OpenMRS[47]. For the comparison, I used free trial versions of SonarQube and PMD. By digging into the tool's documentation, I first tried to group similar rulesets and then created rulesets for each tool. Table 3.4 shows the names of rules and the number of rule violations detected by each tool.

Rule name	Number of violated rules
PMD Detected Rules	#

Rule name	Number of violated rules
CognitiveComplexity	181
GodClass	50
CyclomaticComplexity	258
TooManyMethods	49
LawOfDemeter	296
CouplingBetweenObjects	48
ExcessivePublicCount	40
NcssCount	48
SwitchStmtsShouldHaveDefault	4
TooManyStaticImports	13
SonarQube detected Rules	#
The cognitive complexity of methods should not be too high	196
"Switch case" clauses should not have too many line of code	3
"Switch" statements should have "default cases"	4
Classes should not have too many "static" imports	10
Cyclomatic complexity of the method should not be too high	138
Methods should not be too complex	30
The two branches of a conditional statement should not share an identical structure.	12

Table 3.4. *Issues detected by PMD and SonarQube*

Following the analysis of results from both tools, PMD produced its output as an HTML report, categorizing all detected rule violations as code smells. In total, PMD identified 987 rule violations. The report provided detailed information about each violated rule, including the name of the class where the violation occurred and the corresponding line number. Conversely, SonarQube identified 439 code smells, 41 bugs, 0 vulnerabilities, 26 security reviews, and 542 duplicated blocks. SonarQube presented its findings on an interactive dashboard, which further classified code smells based on severity levels such

as blocker, critical, major, minor, and informational. The SonarQube dashboard report included information on the rule violations, the classes and methods affected, the effort required to refactor the code, and the type of rule violations.

To ensure consistency, similar rules were configured in both tools, including thresholds for cyclomatic complexity, cognitive complexity, a maximum number of methods allowed in a class, and switch-case structures. Despite these identical configurations, the results varied significantly between the tools. For instance, with a cyclomatic complexity threshold of 10, PMD detected 258 violations compared to 138 identified by SonarQube. Similarly, for cognitive complexity, with a threshold of 15, discrepancies in detection persisted. When the threshold for the maximum number of methods in a class was set to 20, PMD identified 49 violations, whereas SonarQube reported 30. These inconsistencies can primarily be attributed to differences in how each tool interprets and applies the rules, as highlighted by Lenarduzzi et al.[40]. The low precision of these tools often stems from the high sensitivity of their rule configurations, where overly strict threshold values can result in numerous false positives. Silva et al. (2018) [50] further emphasize that most static analysis tools suffer from significant false positive rates, with precision levels ranging in between 18% and 57%

3.2.4 Discussion and Reflection

The findings of this study offer valuable insights for researchers and tool vendors to guide their selection of the most suitable static analysis tool (SAT) for specific projects. By comparing two widely used SATs—PMD and SonarQube—based on evaluation factors and their ability to detect predefined static analysis rules, this study sheds light on the strengths and weaknesses of each tool. Using a Java project hosted on GitHub as a case study, I derived a set of similar rules for detection by both tools, further comparing their detection results at the line and class levels. Additionally, a manual precision analysis was conducted to evaluate their accuracy in identifying quality issues.

One of the key takeaways from this study is that there is no "silver bullet" among static analysis tools when it comes to source code quality assessment. Both tools analyzed in this study—PMD and SonarQube—demonstrated their ability to detect different types of source code quality issues, but neither comprehensively covered all potential issues. PMD excelled in identifying certain rule violations, such as cyclomatic complexity, while SonarQube provided a broader assessment of issues, including security vulnerabilities and duplicated blocks. This disparity in detection capabilities highlights that each tool is tailored to address specific quality concerns, making it necessary for teams to align their tool selection with their unique project requirements.

From a practical perspective, this finding suggests that software development teams aiming for robust source code quality assessment may need to combine multiple SATs. By leveraging complementary tools, practitioners can gain a more holistic view of quality is-

sues and technical debt within their codebase. However, combining tools also introduces challenges, such as managing overlapping or conflicting rule detections and the increased effort required to configure and interpret results.

The results of this study emphasize that there is no universally best tool that fits all use cases and organizational needs. Each tool has specific strengths and weaknesses, making it essential for teams to carefully evaluate their priorities and project goals when selecting a tool. For instance, SonarQube offers advanced features such as technical debt calculation, security vulnerability detection, and customizable dashboards, making it suitable for projects requiring detailed quality metrics and ongoing monitoring. On the other hand, PMD provides a lightweight, customizable ruleset engine that is particularly effective for teams focusing on coding style and maintainability.

Moreover, the way tools calculate and report quality metrics, such as technical debt principal and interest, varies significantly. As shown in this study, the two tools often disagreed in their detection of similar rule violations, demonstrating that interpretation differences can impact results. Teams should therefore prioritize tools that align with their coding practices and quality objectives, ensuring that the tool's rule engine and metrics provide actionable insights relevant to their context.

4. CASE STUDIES ON TECHNICAL DEBT MANAGEMENT

4.1 Research design and approach

To answer RQ2 this study adopts a quantitative research approach through the use of a systematic review methodology[51]. The purpose is to conduct a systematic review of existing case studies that focus on the management of technical debt (TD) within organizations. These case studies typically involve interviews with, and data collection from, key stakeholders in organizations that have been actively managing technical debt. The research aims to derive insights into the most effective strategies for managing technical debt from an organizational perspective by analyzing the methods and processes identified in these studies.

The rationale for using systematic review as the primary research methodology lies in its structured, transparent, and replicable approach[52]. Systematic reviews allow for the comprehensive collection, critical evaluation, and synthesis of existing research, ensuring that all relevant studies are included, and the analysis is conducted objectively[52]. In this context, a systematic review provides the best approach to evaluating how technical debt is managed across different organizations and identifying common strategies that lead to successful outcomes.

In this study, the focus is on the systematic evaluation of existing case studies from a broad, organizational point of view. The goal is to identify patterns across multiple cases rather than delve into the specific experiences of individual stakeholders. Therefore, a quantitative approach through systematic review is more appropriate for synthesizing findings from different organizations and providing generalized conclusions about effective technical debt management practices.

This study evaluates case studies that concentrate on how various processes and practices are used in technical debt management in organizations. The organizational perspective is essential because technical debt is not solely a technical issue but also an organizational challenge that requires coordination across different teams and management levels. By focusing on case studies that examine technical debt from this perspective, this research aims to identify methodologies that are not only technically effective but also align with organizational goals and workflows. The ultimate aim is to provide insights that can help organizations optimize their approaches to technical debt management in a

way that improves long-term software sustainability and business performance.

4.2 Research process

The research process followed in this study is designed to rigorously and systematically collect, evaluate, and synthesize data from existing case studies on technical debt management. Given the systematic review methodology employed, the process is divided into several key stages, each ensuring that the review is comprehensive, transparent, and replicable.

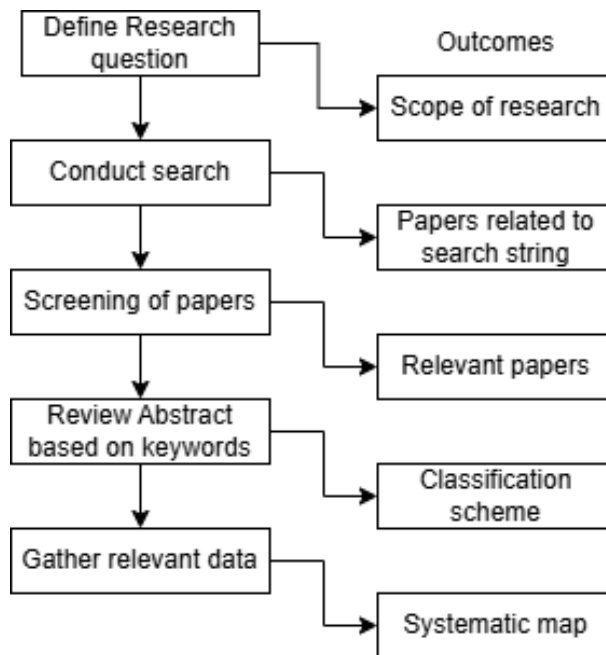


Figure 4.1. Research process in answering RQ2

The first step in the research process involved the formulation of research question and the definition of clear inclusion and exclusion criteria for selecting relevant case studies, The research question is RQ2: How can an organization manage technical debt successfully? The sub-research question is What are the activities performed in an organization to successfully manage the TD? This research question was designed to guide the review towards understanding which methodologies and strategies are most effective for managing the technical debt within organizations. The inclusion criteria were established to ensure that only case studies focusing on organizational practices in technical debt management were considered. The exclusion criteria was to exclude the research papers and studies that do not mention the list of TDM activities specified by Li et al.[27].

Following the definition of research questions and criteria, a literature search was conducted across multiple academic databases, including Scopus, IEEE Xplore, Science direct, Research Gate and Tuni Library (Andor). The search was designed to capture a wide range of case studies on technical debt management, using search terms such as

"technical debt management," "software debt," "technical debt strategies," and "organizational practices in technical debt.". Finally I used a searched using this search string to maintain uniformity through the search "(("technical Debt management") AND ("Organization" OR "Industry"))". The number of results obtained from each database is highlighted in the figure 3.1. Other than search string I applied the filters for including papers only in English, papers only after 2015 and papers related to software industry.

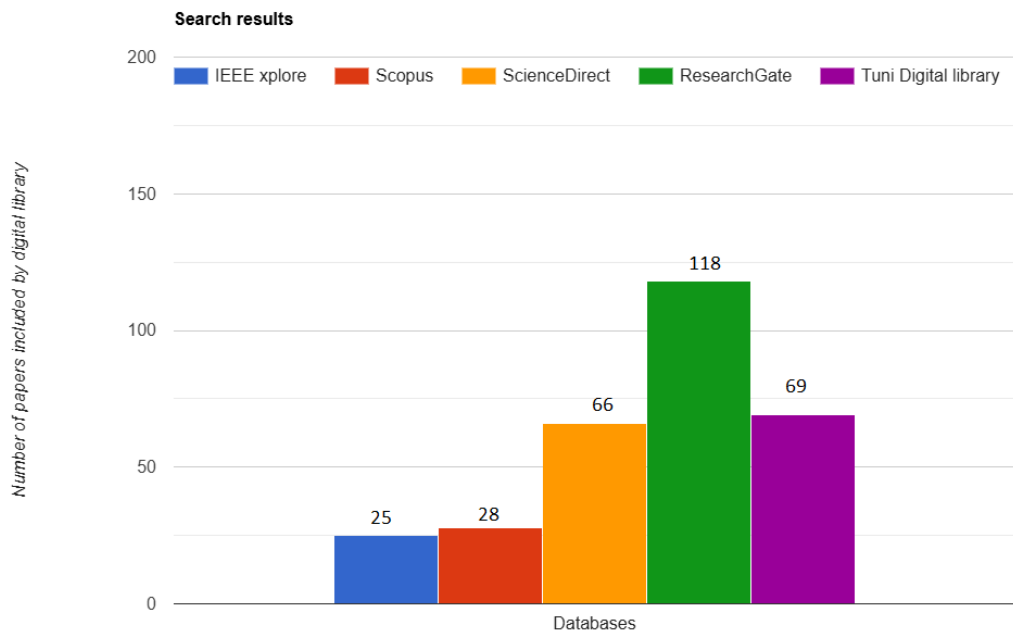


Figure 4.2. Number of papers across various databases

After the literature search, the next stage of the process was the screening and selection of studies. Initially, the titles and abstracts of all identified studies were reviewed to eliminate those that did not meet the inclusion criteria. The studies that cleared the initial screening were subsequently subjected to a full-text review, during which their methodologies, findings, and relevance to the research questions were thoroughly assessed. This two-step screening process helped to ensure that only high-quality and relevant studies were included in the systematic review. I only included the papers with technical debt management activities studied in [27]

In total, I selected 7 such case studies that match my inclusion and exclusion criteria. Then a data extraction process was implemented. A standardized data extraction form was created to capture relevant information from each case study, including details on the organization being studied, the technical debt management strategies used, the outcomes of these strategies, and any challenges or limitations encountered. This structured approach to data extraction ensured consistency across studies and facilitated the synthesis of findings. Table 4.2 represents the case studies that I selected for data extraction

process.

Case	Title
Case 1 [50]	Technologies to support the technical debt management in software projects: a qualitative research. (2018).
Case 2 [22]	An enterprise perspective on technical debt
Case 3 [53]	Managing technical debt: An industrial case study
Case 4 [54]	How do software development teams manage technical debt? – An empirical study
Case 5 [55]	Measure it? Manage it? Ignore it? software practitioners and technical debt.
Case 6 [56]	Success Stories in Managing and Paying Down Tech Debt
Case 7 [57]	Technical Debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations.

Table 4.2. Selected case studies

The final stage of the research process involved the synthesis of results. The extracted data were analyzed to identify common methodologies and practices used across different organizations. This synthesis allowed for the identification of patterns and trends in technical debt management, which were then used to answer the research questions. In particular, the analysis focused on identifying which strategies were most commonly associated with successful technical debt management and how these strategies were implemented within organizations. The synthesis process also involved critically evaluating the findings from the different case studies to ensure that the conclusions drawn were robust and well-supported by the evidence.

The research process concludes by formulating recommendations and conclusions based on the synthesized findings. These conclusions are presented with an emphasis on providing actionable insights for organizations seeking to manage their technical debt effectively.

4.3 Case studies

Silva, Victor Machado da[50] in his qualitative research about technology support in TDM, surveyed Brazilian software organizations to gather information on existing methods used for TDM. The authors carried out the survey on a group of practitioners working at different Brazilian software organizations. The authors designed the questionnaire in such a way as to understand the existing knowledge about TD among the participants and if the participants had any experience of effective TDM activities in their recent software projects. The practitioners who participated in the survey served different roles in Brazilian software organizations. The participants were asked to select the TDM activities they had observed in their organization or the activities they were part of. The background for this TDM activities was given to the participants to avoid any confusion. The list of these TDM activities was derived from [27]. In total, 67 practitioners from different education levels and various roles in the software organization, participated in the survey. Out of 67 participants, 40 answered the survey completely. Out of these 40 only 24 were aware of the term technical debt. 9 answered about the TDM activities at their organization. The most common TDM activity was TD identification and TD documentation. TD Communication was also among the commonly conducted TDM activity. The responses showed varied approaches across identification, documentation, communication, measurement, prioritization, repayment, and prevention of TD. For TD identification, two participants recognized a formal, necessary strategy, one acknowledged a non-mandatory strategy, and three reported no formal strategy. TD identification was described as a continuous process by three participants, with others identifying TD only when issues arose, and classifications varied, including categories like Design and Documentation Debt, origin artifact, and “effort vs. business value” considerations. Regarding TD documentation, two participants followed a standardized documentation method, with one reporting a non-mandatory standard and two using informal methods. Four participants used a general backlog for documenting TD, while one used a specific TD backlog. In TD communication, four participants discussed TD in project meetings with limited stakeholder involvement, while one reported only informal discussions. TD measurement saw minimal engagement, with only two participants responding. One measured TD informally using metrics, while the other used a formal approach with item count. Information used for measurement included “effort vs. value,” person-hours/months, and lines of code (LOC). For TD prioritization, guesses or experience-based estimates were common, with considerations for client impact, project impact, severity, system-wide usage, and cost/benefit ratios among the prioritization criteria. TD repayment was either planned according to project needs, continuously, or undertaken as a last resort. Finally, TD prevention was informally managed by individual team members without a formal process.

Klinger et al. [22] conducted interviews with 4 architects at IBM to get a perspective on TDM in different kinds of projects. The four architects who participated in the survey

were referred to as Subjects A, B, C, and D respectively. Subject A was responsible for a relatively new offering that integrated into an established product portfolio at IBM. This solution needed to interface with current and emerging technologies. Subject B, an experienced architect. Subject C is an architect who successfully integrated the acquired product into existing IBM products. Subject D is also an experienced architect who worked on large-scale projects in IBM. Klinger et al. conducted semi-structured interviews with the subjects. Each subject faced different encounters with tackling the TD. Subject A had to choose between designing a product to work with an older API that would soon be outdated or a newer API that was expected to be used in future versions. Both options would work for the current project, but only the new API would meet long-term needs. But using the new API required help from another team and the new team had another priority task and only the older API was compatible with other components. Because the partner team didn't support the new API, Subject A's team faced "technical debt," meaning that choosing the old API would create problems they'd have to fix later. Subject A felt that this technical debt could have been avoided with better communication and estimation. Subject B joined to work on a new version of a product that IBM recently acquired. Before IBM's acquisition, some requirements were given low priority because they weren't crucial for the original customers. However, after the acquisition, IBM's global reach made these previously low-priority needs—like accessibility, globalization, and performance—much more important, especially as the product became part of a larger software suite. Alongside these requirements, the team also had to add new features. This shift led to "technical debt" because the product's original design didn't account for these newly important requirements. Subject B felt that this TD could have been avoided with better TD prioritization. Subject C's company was acquired by IBM, and he was tasked with integrating his product with an existing IBM product. Although both teams agreed on the ideal integration approach, the existing product team had limited resources and prioritized delivering promised features over creating new APIs for integration. As a result, they chose an older, less optimal integration method. While this decision met immediate needs, it introduced technical debt for both teams, as they would eventually need to replace the stop-gap solution with a better integration method. Subject D shared insights from projects within and outside IBM, highlighting a non-IBM desktop product that became very successful but accumulated technical debt. A team was formed to rewrite it from scratch to reduce this debt, but they overlooked many ways customers depended on specific quirks of the original product. Without understanding these hidden requirements, they couldn't fully address the debt. The rewrite attempt risked alienating customers, leading the company to cancel it. This case illustrates how hard it can be to identify and manage technical debt, as it requires a deep understanding of both the technology and the ecosystem around it. Overall, from these interviews, it is clear that the above incurred technical debt could be managed with better visibility, communication and prioritization.

Santos et al. [58] studied the TDM with their industrial partner and aimed to find the best practices and known challenges for TDM in an industry adopting agile development practices. 28 personnel participating in the study worked in different roles such as product owners, scrum masters and team members. The interview questions were divided into three parts. The first part covered demographic information. The second part explored how Scrum influenced their projects and their views on Agile adoption. The third part consisted of general questions about managing technical debt. Based on the survey results of the third part, the most common debt were code, testing and architectural debt. As per the participants, the most difficult TD to manage would be architectural debt because this type of debt requires the most efficient communication among different teams to achieve refactoring. The answers from participants showed that the debt such as code debt and testing debt were managed by a separate team that detects and solves the debt continuously. In addition to the TD reduction teams other teams also work on TDM by writing automation and manual test cases. Overall, the study conducted in [58] showed that effective TDM requires continuous detection and resolution. However, for managing TD such as architectural TD efficient communication among teams is required.

Yli-Huumo et al.[54] performed a case study on a large software organization to study the successful implementation of TDM strategies. The study's main aim was to gather empirical evidence related to TDM in a large software organization. The authors conducted interviews with 25 participants from 8 different software teams. The study identified the teams that effectively manage TD and the techniques used in those teams. The authors also identified the challenges in TDM. [54] considered 8 software teams as cases: Case A to Case H. The selected case company was a software supplier company with over 5600 employees. The authors conducted semi-structured interviews with 25 participants. To analyze the gathered data, the authors conducted a mapping study based on Li et al.'s [27] eight activities of TDM TD repayment, TD representation/documentation, TD identification, TD prioritization, TD measurement, TD monitoring, TD communication, and TD prevention. The analysis revealed that TDM activities were performed at varying levels of maturity. For instance, one development team prioritized and actively engaged in measurement and monitoring tasks, while another team allocated no effort to these activities. Three TDM maturity levels were identified: unorganized, received, and organized. An unorganized TDM activity occurs when a software team either neglects the activity or gives it minimal focus. A received TDM activity is one where the team has acknowledged its importance and has begun to implement it, albeit on a limited scale. At this stage, the activity is not consistently practiced and is only performed sporadically by a few members. Communication was the most common TDM activity among the development teams studied. Technical Debt (TD) was frequently discussed, but a communication gap often existed between technical and non-technical stakeholders. This gap hindered the resolution of TD issues, as business stakeholders were not adequately informed. Effective TD commu-

nication is crucial for TDM success, and most teams had organized communication well, aiding other TDM activities. Occasionally performed activities include TD repayment, TD presentation, TD documentation, TD identification. TD prevention is considered one of the most impactful activities among the eight TDM practices a development team can implement. By establishing mandatory coding standards, supported through practices like code reviews and a clear Definition of Done, the amount of TD introduced into the codebase can potentially be reduced. Minimizing TD upfront also benefits other TDM activities. Moreover, implementing TD prevention measures is particularly useful for identifying and addressing less optimal solutions from inexperienced developers. TD identification was carried out by the development team both manually and using tools. Architects and developers who did not use any tools stated that the tools are vital to identify TD caused by architectural and structural issues. Based on the responses, the authors also observed the known challenges in TDM. The main challenge is not using enough TDM tools. They also noticed that TDM requires significant time and effort. Introducing new TD processes and tools can add extra workload to the existing development activities. Other major challenge in TDM today is prioritizing TD. The difficulty lies in the absence of effective models and methods to prioritize TD issues effectively. There are no adequate solutions to help understand or articulate why certain TD items should take precedence over others for the development team.

Ernst et al. [55] investigated which tools and TDM practices were adopted by a software organization to manage TD. The study found that 65% of the total participants did not carry out any TDM activity. While 25% of participants perform TDM activities but not in any organized manner. 10% percent of participants said that only team managers were performing TDM activities. As per the participants, the most common debt was architectural debt. However, no specific tools were being used for identifying the architectural debt because of the complexity involved in setting up the tool. Some teams used the tools for TD communication or tracking and TD identification. Ernst et al. recommend that research on technical debt tooling emphasize monitoring the gap between development and architecture, while enhancing ongoing architecture analysis and ensuring conformance. Carlos Fuentes [56] in success stories about managing technical debt highlights the importance of having a strategy for managing technical debt as a technical leader on a project. He emphasizes the value of maintaining a backlog dedicated to technical debt tasks and communicating this backlog with key team members and stakeholders—including project managers, business analysts, developers, and other technical leads—to identify high-priority items that need scheduling in upcoming sprints. By making technical debt visible and manageable, and aligning it with the available developer capacity, teams can address technical debt incrementally. He recommends aiming to tackle one or two technical debt tasks per sprint, which allows the debt to be gradually reduced. In conclusion, Carlos's approach underlines that systematic planning, prioritization, and clear communication among stakeholders are essential for managing and progressively

reducing technical debt within development cycles.

Martini et al. [57] conducted a survey to examine technical debt (TD) management practices in 15 software organizations. The study involved interviews with 226 participants occupying various roles in the software industry, including developers, architects, and managers. The primary objectives of the research were to analyze the cost of managing TD, identify effective tools for tracking TD, and investigate the TD management processes within these 15 Scandinavian organizations. The interviews were structured into three distinct sections. The first section gathered information about the participants' profiles, professional experiences, and related details. The second section focused on questions regarding TD management processes and the perceived importance of TD management. Finally, the third section explored the outcomes of implemented TD management practices. Analysis of the survey results revealed that, on average, 25% of total time was dedicated to TD management activities. Developers were found to be the most familiar with tools and practices related to TD management. Among the methods for tracking TD, using a backlog emerged as the most common approach, while static analysis tools were particularly effective in mitigating code-related debt. Despite these findings, the study highlighted a significant lack of awareness about TD management tools, with only 27% of participants actively using such tools to track TD. The authors emphasized that this lack of awareness could exacerbate the negative effects of TD. They also noted that communication, backlog management, and the use of static analysis tools were the most frequently implemented practices for managing TD.

4.4 Results

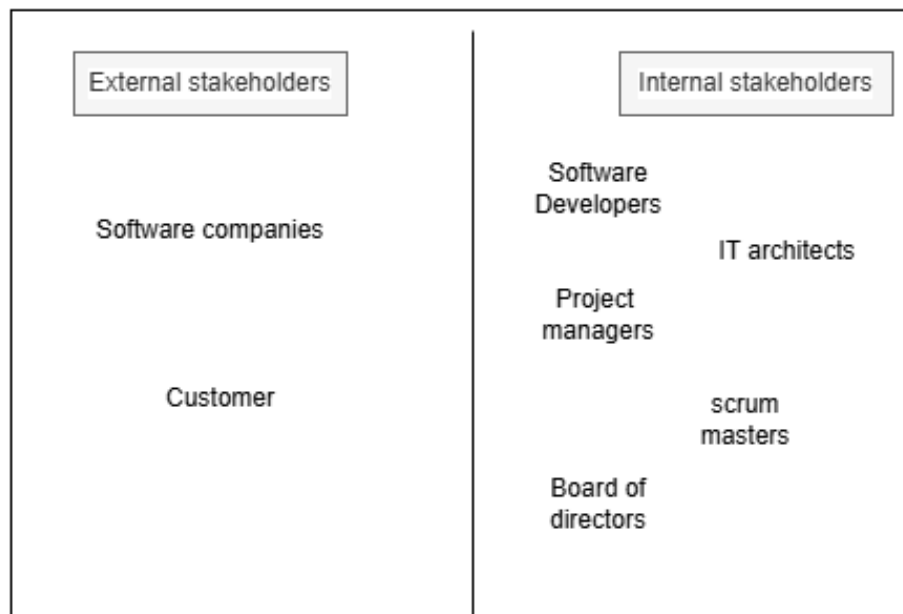


Figure 4.3. Stakeholders involved in TDM

After examining the seven case studies and data, we have discovered that numerous stakeholders participate in technical debt management practices, as shown in Figure 4.3. Technical debt is managed prominently by internal and external stakeholders, such as project managers, developers, and other domain experts. Internal stakeholders, such as developers, are primarily responsible for identifying and managing technical debt during the software development lifecycle. Meanwhile, external stakeholders can assist in prioritizing and resolving technical debt to ensure the long-term success of the software project.

In this study, seven cases from software organizations were analyzed to understand the adoption and implementation of Technical Debt Management (TDM) activities. These TDM activities are outlined by Li et al.[27] and they are identification, measurement, prioritization, prevention, monitoring, repayment, representation, and communication. The investigation of seven cases revealed that certain TDM activities are more prevalent and widely implemented across organizations. Specifically, Communication, Identification, Monitoring, Prioritization, and Repayment were identified as the most popular and frequently practiced activities. The Table 4.3 summarizes the implementation of TDM activities across the seven studied cases. Each row represents a specific case, and the columns correspond to different TDM activities: Identification, Measurement, Prioritization, Monitoring, Communication, Repayment, and Prevention. A checkmark in a cell indicates that the corresponding activity was implemented in the respective case. From the Table, it is evident that Communication, Identification, Monitoring, Prioritization, and Repayment are the most commonly implemented activities, as they appear across multiple cases. Measurement and Prevention, while important, are less frequently adopted, suggesting that some organizations focus more on addressing existing debt rather than quantifying or preventing it.

Cases	Identification	Measurement	Prioritization	Monitoring	Communication	Repayment	Prevention
Case 1	✓		✓		✓	✓	✓
Case 2	✓		✓		✓		
Case 3	✓				✓	✓	
Case 4	✓			✓	✓	✓	
Case 5	✓				✓		
Case 6			✓	✓	✓		
Case 7	✓			✓	✓	✓	

Table 4.3. TDM Activities Across Cases

From the studied cases, it was evident that tools supporting Technical Debt Management (TDM) activities are neither sufficiently explored nor widely adopted. A significant factor contributing to this gap is the lack of awareness among practitioners about the existence and capabilities of these tools. Many organizations rely on manual processes or ad hoc methods for managing technical debt, which limits the efficiency and effectiveness of their

TDM practices. To address this, I systematically introduce the list of activities and tools from the existing studies that have examined these strategies and tools. This mapping will help researchers and practitioners to get familiar with the approaches and tools that can enhance technical debt management strategies. By leveraging the right tools, organizations can streamline TDM processes, reduce overhead, and improve long-term software quality and maintainability[6].

The Table 4.4 on Communication highlights the key activities and approaches organizations can adopt to ensure effective collaboration among stakeholders during the technical debt management (TDM) process. The table also outlines the tools that supports the approaches such as TD dashboard, backlog management, and dependency visualization. The last column cites studies or research papers that have examined these communication strategies and tools in TDM contexts, providing academic evidence for their effectiveness.

Table 4.4. *Technologies and tools for TD Communication*

TDM activity	Technology and description	Tools	Studies
TD communication	<ul style="list-style-type: none"> • TD dashboard - A dashboard shows TD items, categories, and quantities to ensure all stakeholders are aware of the TD's presence. • Backlog - All identified TD items, along with any issues to be addressed during development, are added to the software project's backlog • Dependency Visualization - Display the unwanted dependencies between software components. 	GitHub, Debt-Flag, BitBucket, Jira, Asana	Yli-hummo et al.,2016[54], Dos Santos et al., 2013[59]

The table 4.5 focuses on the strategies and tools used to detect and categorize technical

debt. The first column presents the technologies or methodologies employed, including code analysis, Dependency analysis, architectural visualization, etc. The table introduces tools which are widely used to identify specific types of technical debt. The fourth column references studies that validate these tools and approaches, showcasing their practical and theoretical relevance in TD identification.

Table 4.5. Technologies and tools for Identification

TDM activity	Technology and description	Tools	Studies
TD identification	<ul style="list-style-type: none"> • Code analysis - Examine source code to detect coding rule violations, missing tests, and design or architectural problems. • Dependency analysis - Examine the relationships between various types of software elements. • Architectural smell detection - Analysis of software architecture and detects architecture violations and architectural smells that may indicate poor code structure. 	SonarQube, SonarQube COBOL plugin, CLIO, CodeVizard, FindBugs, PMD, PHPMD, NDepend, NCover, FxCop, CodeXpert, Cobertura, Checkstyle.	Yli-hummo et al.,2016[54], Zazworka et al.,2013[29], Ernst et al., 2015[55]

The table 4.6 on Prioritization outlines methods and tools for ranking technical debt items based on their impact and urgency. This describes the approaches, such as Cost/benefit analysis, TD interest that guide prioritization efforts.

Table 4.6. Technologies and tools for Prioritization

TDM activity	Technology and description	Tools	Studies
TD prioritization	<ul style="list-style-type: none"> • Cost/benefit analysis - If addressing a TD item provides benefits that outweigh the costs, it should be resolved. TD items with the highest benefit-to-cost ratios should be prioritized for repayment. • Prioritization based on TD interest - TD items that accumulate higher interest should be prioritized for repayment. 	Sonar TD plugin, (SQALE) plugin for SonarQube, JSpIRIT	Yli-hummo et al.,2016[54]

The Repayment table 4.7 addresses the strategies and tools used to resolve or mitigate technical debt. The second column details the approaches, like incremental refactoring or automated testing frameworks, that enable TD repayment. According to [27] refactoring is most studied TD repayment approach. In refactoring, code, design and architecture is altered to improve internal quality while preserving external behavior. Other approaches mentioned are automation, repackaging, re-engineering, rewriting, bug fixing, and fault tolerance.

Table 4.7. *Technologies and tools for Repayment*

TDM activity	Technology and description	Tools	Studies
TD repayment	<ul style="list-style-type: none"> • Refactoring - Modify the code, design, or architecture of a software system to enhance its internal quality, while preserving its external functionality. • Automation - Automate processes such as manual test, manual deployment, and builds. • Bug Fixing - Resolve known TD manually • Repackaging - Group-related modules with controllable dependencies to make the code simpler. 	Sonar TD plugin, Automation testing tools, CI/CD tools	Yli-hummo et al.,2016[54]

The Monitoring table 4.8 emphasizes the ongoing activities required to track and control technical debt over time. This outlines monitoring activities, such as the Definition of Done which is a threshold-based approach, in this approach thresholds for TD quality metrics are defined, issuing warnings when not met. Other approaches are TD tracking and Quality checks which are supported by tools such as Software maps tools and SonarQube.

Table 4.8. *Technologies and tools for Monitoring*

TDM activity	Technology and description	Tools	Studies
TD Monitoring	<ul style="list-style-type: none"> • DoD(Definition of Done) - Establish thresholds for TD related quality metrics and generate warnings when these thresholds are not achieved. • TD tracking - Monitor the impact of TD by examining dependencies between the TD-containing parts of the system and other areas it affects. • Quality checks - Consistently measure identified TD and monitor its progression over time. 	Software maps tool, SonarQube, DebtFlag, Sonar TD plugin	Yli-hummo et al.,2016[54], Oliveira et al.,2015[60]

5. DISCUSSION

This chapter synthesizes the findings of the study, contextualizing them within the broader landscape of Technical Debt Management (TDM). The research addressed two key questions: How to identify technical debt using tools? and How can an organization manage technical debt successfully? By analyzing case studies, existing literature, and tool evaluations, this thesis provides insights into the current state of TDM practices, their challenges, and strategies for improvement.

5.1 Technical Debt Identification Using Tools

Technical Debt Identification Using Tools is the answer to RQ1 regarding how to detect technical debt such as code, design, architecture and test debt. We examined ten recognized types of technical debt (TD). However, this study primarily focused on four key categories: code debt, design debt, architecture debt, and test debt. Because as per Alves et al.[2], Ampatzoglou et al.[26], and Parthiban[18] these are the most commonly occurring debt in software environments. Table 2.1 provides an overview of these technical debt types, along with specific items or indicators that contribute to the accumulation of each type. The study highlights the underutilization of tools for identifying technical debt, despite the availability of numerous options in the market. The lack of awareness among practitioners emerges as a significant barrier, often leading to reliance on manual processes or ad hoc methods. Another key finding is the difficulty in selecting the right tool for specific project needs. Practitioners face challenges due to the diverse nature of technical debt, which spans code smells, architectural issues, and other technical debt indicators. This research offers a categorization of tools based on their functionality and usability in Table 3.1. The tools in static analysis tool identifies the code debt because this tool specializes in identifying TD indicators such as code smells, code guideline violations etc. Architecture analysis tools are used to detect design and architecture debt which is caused by design flaws and architecture smells. Test management tools help prevent the quality issues caused by inefficient test cases. But it is also important to choose right tool for project needs [40][38]. I described the evaluation criteria for selecting a tool such as technology and languages, features, integration capabilities, costing, and the tool's capability to provide insights. The detailed description of tools in each category based on the evaluation criteria shows that the tools are effective way to early identification of TD. The evaluation criteria served as the foundation for comparing PMD and SonarQube the

two widely used static analysis tools [41]. The results highlighted that neither tool was universally superior; instead, each addressed specific organizational needs. PMD demonstrated strength in providing targeted analysis of rule violations and was particularly effective for detecting cyclomatic complexity and other code quality issues. Its lightweight and open-source nature makes it an attractive option for teams with limited budgets or narrow technical requirements. SonarQube, with its broader range of features and multi-language support, was more suitable for teams requiring comprehensive quality assessments, including security vulnerabilities, duplicated blocks, and overall maintainability. Its advanced integration capabilities make it ideal for organizations seeking to embed quality checks seamlessly into their workflows. Practitioners should prioritize tools that best align with their project's technology stack, quality goals, and budgetary constraints. The evaluation criteria outlined in this study can serve as a checklist to guide this process.

5.2 Effective Management of Technical Debt

Effective management of technical debt answers our second research question RQ2. Chapter 4 presents an analysis based on a quantitative research approach, examining seven case studies from software organizations. Through this analysis, we identified the strategies these organizations applied to manage technical debt (TD) effectively. By examining these cases, we were able to identify the key stakeholders involved in technical debt management (TDM) and clarify the specific roles and responsibilities of each stakeholder within TDM processes.

Based on these findings, we created tables that aligns TDM activities with relevant technologies that facilitate their execution. Additionally, we highlighted tools that specifically support each TDM activity, providing organizations with a resource to enhance their TD management practices.

In summary, organizations can successfully manage technical debt by following a structured approach, which includes identifying responsible stakeholders, employing supportive tools, and adhering to targeted TDM activities. The following steps, when systematically implemented, contribute to effective TD management and help maintain the long-term health and efficiency of the software system.

Communication: Strong communication between developers, architects, and managers is critical for aligning efforts in managing technical debt. This activity is widely adopted, as it ensures that all stakeholders have a shared understanding of debt-related issues and mitigation strategies.

Identification: Identification is the first step in managing TD and involves detecting, categorizing, and documenting debt across various dimensions, such as code quality, architecture, and testing. This activity lays the groundwork for all subsequent TDM efforts. Approaches like static code analysis, architectural reviews, and test coverage evaluations are commonly used for identifying TD. Tools such as SonarQube, PMD, and ArchUnit

assist in detecting issues related to maintainability, complexity, and design flaws.

Prioritization: Once TD is identified, prioritization is critical for determining which debt items should be addressed first based on their impact and urgency. Effective prioritization considers factors such as the severity of the debt, its impact on the system, and the cost of repayment. Techniques like cost-benefit analysis, risk assessment, and weighted scoring models help teams decide which debts to tackle first.

Repayment: Repayment involves resolving or mitigating identified technical debt through targeted actions such as refactoring, redesigning, or automating tests. This activity requires strategic planning to balance debt repayment with ongoing development tasks.

Monitoring: Monitoring is a continuous activity aimed at tracking the status of technical debt over time. It includes evaluating metrics like code quality, maintainability, and debt repayment progress to prevent unchecked accumulation of technical debt.

5.3 Threats to Validity

This section examines four potential threats to the validity of the study, focusing on four key dimensions: construct validity, internal validity, external validity, and conclusion validity suggested by [61]. Each dimension is discussed in detail below.

5.3.1 Construct Validity

Construct validity relates to the degree to which the research accurately captures the concepts it seeks to investigate. In this study, the constructs of technical debt (TD) and technical debt management (TDM) activities were central. However, TD is a multifaceted concept, encompassing aspects like code quality, architectural issues, and even documentation gaps. The broad scope and varied interpretations of TD could lead to inconsistencies in how it is understood and applied across different contexts. Furthermore, the study's comparison of PMD and SonarQube was limited to their respective feature sets, which may not comprehensively represent the full landscape of static analysis tools available. Additionally, the surveys and interviews conducted may not have captured all relevant aspects of TDM activities, as the design of these instruments inherently limits the range of responses. To address these challenges, the study relied on established frameworks and triangulated findings across multiple data sources, ensuring that key dimensions of TDM were consistently represented.

5.3.2 Internal Validity

Internal validity means the factors that may have influenced the study. Participant variability in case studies or interviews poses a threat, as individuals' roles, expertise levels, and organizational practices may lead to diverse perspectives on TDM. Confounding vari-

ables, such as the impact of organizational culture, team size, or resource availability, might also have shaped the outcomes but were not explicitly accounted for.

5.3.3 External Validity

External validity are related to the generalization of findings. While the study analyzed seven case studies, the limited sample size might not fully represent the diversity of software organizations. Factors such as organizational scale introduce variability that this study does not capture. Moreover, the tool comparison focused exclusively on PMD and SonarQube, leaving out other tools that may perform better in different environments or for specific requirements.

5.3.4 Conclusion Validity

Conclusion validity relates to the credibility and reliability of the interpretations drawn from the study. The limited number of case studies and tools analyzed could constrain the statistical significance of the results, making it challenging to generalize findings with high confidence. Furthermore, the qualitative nature of the research, including interviews and case studies, introduces an element of subjectivity in the interpretation of data. Researchers may unconsciously emphasize specific findings or patterns, thereby influencing the conclusions.

5.4 Future Work

Exploration of Additional Tools: The study focused on PMD and SonarQube as representative static analysis tools (SATs). Future work could expand this scope to include a wider variety of tools, such as Checkstyle, FindBugs, and ESLint, as well as tools that specialize in domain-specific or architectural TD. This broader evaluation could help practitioners identify the most suitable tools for specific use cases.

Integration of AI-Powered Tools for TDM: The incorporation of AI tools into TDM represents a promising avenue for future research. AI and machine learning (ML) can enhance TD identification, prioritization, and repayment by detecting patterns in codebases, predicting future TD based on historical data, and automating decision-making processes.

Longitudinal Case Studies: This thesis captured TDM practices through cross-sectional case studies, offering a snapshot of organizational approaches. Conducting longitudinal studies would enable researchers to observe how TDM practices evolve over time and assess their long-term effectiveness, including the sustained impact of tool adoption and best practices.

Economic Impact of TDM: Quantifying the economic benefits of effective TDM practices could strengthen the case for organizational investment in these activities. Future work could conduct cost-benefit analyses to measure the financial impact of reduced TD on software maintenance, development efficiency, and overall project outcomes.

Developing Standardized Metrics: A key challenge identified in this research is the lack of standardized metrics for measuring and tracking TD. Future work could aim to develop universal frameworks and benchmarks, enabling better comparison of TD severity and management practices across projects and organizations

6. CONCLUSION

This thesis has explored the critical challenges and practices associated with managing technical debt (TD) in software organizations. By addressing two central research questions—how to identify TD using tools and how organizations can effectively manage it—this study provides valuable insights into the tools, methodologies, and best practices for technical debt management (TDM).

The findings underscore the importance of systematic TD identification as a foundation for successful TDM. This study focuses on understanding technical debt and the tools used for identifying it. The most commonly detected technical debt are Code TD, Design TD, Architecture TD and Test TD[3], [62], [18]. Through a literature review, we derived 3 distinct categories mainly Static Analysis tools, Architecture analysis tools and Test management tools. These three categories of tools are used to identify the most common types of TD. We evaluated the tools in each category conducted by experts in the field. Based on practical and technological research that we carried out by comparing SonarQube and PMD, none of tool was capable to identify every time of technical debts. The vital factors while choosing the appropriate tool for an organization depends on language support, features of tools, integration capabilities, costing, and detailed analysis of data the tool offers.

In addition to addressing the identification of technical debt, this research investigated best practices for technical debt management (TDM) through a systematic literature review (SLR) of seven selected case studies. The analysis revealed that TDM activities such as communication, monitoring, prioritization, and repayment are commonly implemented across various organizations. However, notable gaps were identified, particularly in the awareness and adoption of advanced tools and methodologies, which could impede effective mitigation of technical debt. To address this, the study compiled a list of tools supporting these TDM activities, derived from an in-depth review of existing research on tool capabilities and applications.

In conclusion, this thesis emphasizes that technical debt is not merely a technical challenge but a multidimensional issue that requires collaboration, strategic planning, and the judicious use of tools and practices. By equipping software teams with the knowledge and tools to effectively manage TD, organizations can enhance software quality, reduce long-term costs, and foster sustainable development practices. This research serves as a foundation for further exploration into this critical aspect of software engineering, driving

improvements in both theory and practice.

REFERENCES

- [1] Cunningham, W. The WyCash portfolio management system. *SIGPLAN OOPS Mess.* 4.2 (Dec. 1992), pp. 29–30. ISSN: 1055-6400. DOI: 10.1145/157710.157715. URL: <https://doi.org/10.1145/157710.157715>.
- [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: A systematic mapping study. *Information and Software Technology* 70 (2016), pp. 100–121. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2015.10.008>. URL: <https://www.sciencedirect.com/science/article/pii/S0950584915001743>.
- [3] Alves, N. S., Mendes, T. S., De Mendonça, M. G., Spínola, R. O., Shull, F. and Seaman, C. Identification and management of technical debt: A systematic mapping study. *Information and Software Technology* 70 (2016), pp. 100–121.
- [4] Biazotto, J. P., Feitosa, D., Avgeriou, P. and Nakagawa, E. Y. Technical debt management automation: State of the art and future perspectives. *eng. Information and software technology* 167 (2024), pp. 107375–. ISSN: 0950-5849.
- [5] Seaman, C. and Guo, Y. Chapter 2 - Measuring and Monitoring Technical Debt. Ed. by M. V. Zelkowitz. Vol. 82. *Advances in Computers*. Elsevier, 2011, pp. 25–46. DOI: <https://doi.org/10.1016/B978-0-12-385512-1.00002-5>. URL: <https://www.sciencedirect.com/science/article/pii/B9780123855121000025>.
- [6] Lenarduzzi, V., Besker, T., Taibi, D., Martini, A. and Fontana, F. A. A systematic literature review on technical debt prioritization: Strategies, processes, factors, and tools. *Journal of Systems and Software* 171 (2021), p. 110827.
- [7] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Lim, E., MacCormack, A., Nord, R., Ozkaya, I. et al. Managing technical debt in software-reliant systems. *Proceedings of the FSE/SDP workshop on Future of software engineering research*. 2010, pp. 47–52.
- [8] Fowler, M. Is design dead?: (2004). URL: <https://www.martinfowler.com/articles/designDead.html>.
- [9] Seaman, C. and Spínola, R. Managing technical debt In:(Short Course) XVII Brazilian Symposium on Software Quality. *SBC, Salvador, Brazil* (2013).
- [10] Novais, R. L., Torres, A., Mendes, T. S., Mendonça, M. and Zazworka, N. Software evolution visualization: A systematic mapping study. *Information and Software Technology* 55.11 (2013), pp. 1860–1883. ISSN: 0950-5849. DOI: <https://doi.org/10.1016/j.infsof.2013.05.008>. URL: <https://www.sciencedirect.com/science/article/pii/S0950584913001298>.

- [11] Guo, Y., Spínola, R. O. and Seaman, C. Exploring the costs of technical debt management—a case study. *Empirical Software Engineering* 21 (2016), pp. 159–182.
- [12] Nord, R. *The Future of Managing Technical Debt*. Carnegie Mellon University, Software Engineering Institute's Insights (blog). Accessed: 2024-Jul-16. Aug. 2016. URL: <https://insights.sei.cmu.edu/blog/the-future-of-managing-technical-debt/>.
- [13] McConnell, S. *Managing Technical Debt [Webinar], Sep. 2011*. 2011.
- [14] Nugroho, A., Visser, J. and Kuipers, T. An empirical model of technical debt and interest. *Proceedings of the 2nd workshop on managing technical debt*. 2011, pp. 1–8.
- [15] Nord, R. L., Ozkaya, I., Kruchten, P. and Gonzalez-Rojas, M. In search of a metric for managing architectural technical debt. *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*. IEEE. 2012, pp. 91–100.
- [16] Li, Z., Avgeriou, P. and Liang, P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), pp. 193–220. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2014.12.027>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121214002854>.
- [17] Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A., Avgeriou, P., Abrahamsson, P., Martini, A., Zdun, U. and Systa, K. The Perception of Technical Debt in the Embedded Systems Domain: An Industrial Case Study. *2016 IEEE 8th International Workshop on Managing Technical Debt (MTD)*. 2016, pp. 9–16. DOI: 10.1109/MTD.2016.8.
- [18] Parthiban, D. G. Examination of tools for managing different dimensions of Technical Debt. *CoRR* abs/1904.11062 (2019). arXiv: 1904.11062. URL: <http://arxiv.org/abs/1904.11062>.
- [19] Maldonado, E. d. S. and Shihab, E. Detecting and quantifying different types of self-admitted technical debt. *2015 IEEE 7th international workshop on managing technical debt (MTD)*. IEEE. 2015, pp. 9–15.
- [20] Detofeno, T., Malucelli, A. and Reinehr, S. Technical Debt Guild: managing technical debt from code up to build. *Journal of Software Engineering Research and Development* (2023), pp. 1–1.
- [21] Banker, R. D., Liang, Y. and Ramasubbu, N. Technical Debt and Firm Performance. *Institute for Operations Research and the Management Sciences* 67.5 (May 2021), pp. 3174–3194. DOI: 10.1287/mnsc.2019.3542. URL: <https://doi.org/10.1287/mnsc.2019.3542>.

- [22] Klinger, T., Tarr, P., Wagstrom, P. and Williams, C. An enterprise perspective on technical debt. (May 2011). DOI: 10.1145/1985362.1985371. URL: <https://doi.org/10.1145/1985362.1985371>.
- [23] Marinescu, R. Assessing technical debt by identifying design flaws in software systems. *IBM* 56.5 (Sept. 2012), 9:1–9:13. DOI: 10.1147/jrd.2012.2204512. URL: <https://doi.org/10.1147/jrd.2012.2204512>.
- [24] Fairley, R. E. and Willshire, M. J. Better Now Than Later: Managing Technical Debt in Systems Development. *IEEE Computer Society* 50.5 (May 2017), pp. 80–87. DOI: 10.1109/mc.2017.124. URL: <https://doi.org/10.1109/mc.2017.124>.
- [25] Ramasubbu, N., Kemerer, C. F. and Woodard, C. J. Managing Technical Debt: Insights from Recent Empirical Evidence. *IEEE Computer Society* 32.2 (Mar. 2015), pp. 22–25. DOI: 10.1109/ms.2015.45. URL: <https://doi.org/10.1109/ms.2015.45>.
- [26] Ampatzoglou, A., Chatzigeorgiou, A., Arvanitou, E. M. and Bibi, S. SDK4ED: A platform for technical debt management. *Software: Practice and Experience* 52.8 (2022), pp. 1879–1902. DOI: <https://doi.org/10.1002/spe.3093>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.3093>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.3093>.
- [27] Li, Z., Avgeriou, P. and Liang, P. A systematic mapping study on technical debt and its management. *Journal of Systems and Software* 101 (2015), pp. 193–220.
- [28] Benldris, M. Investigate, identify and estimate the technical debt: a systematic mapping study. *Available at SSRN 3606172* (2020).
- [29] Zazworka, N., Vetro', A., Izurieta, C., Wong, S., Cai, Y., Seaman, C. and Shull, F. Comparing four approaches for technical debt identification. *Software Quality Journal* 22 (2014), pp. 403–426.
- [30] Ayewah, N. and Pugh, W. The google findbugs fixit. *Proceedings of the 19th international symposium on Software testing and analysis*. 2010, pp. 241–252.
- [31] Guéhéneuc, Y.-G. and Albin-Amiot, H. Using design patterns and constraints to automate the detection and correction of inter-class design defects. *Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39*. IEEE. 2001, pp. 296–305.
- [32] Izurieta, C. and Bieman, J. M. How software designs decay: A pilot study of pattern evolution. *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE. 2007, pp. 449–451.
- [33] Holvitie, J., Licorish, S. A., Spínola, R. O., Hyrnsalmi, S., MacDonell, S. G., Mendes, T. S., Buchan, J. and Leppänen, V. Technical debt and agile software development practices and processes: An industry practitioner survey. *Information and Software Technology* 96 (2018), pp. 141–160.

- [34] Holvitie, J., Leppänen, V. and Hyrynsalmi, S. Technical debt and the effect of agile software development practices on it-an industry practitioner survey. *2014 Sixth International Workshop on Managing Technical Debt*. IEEE. 2014, pp. 35–42.
- [35] Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Arcelli Fontana, F., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., Saarimaki, N., Sas, D. D., Toledo, S. S. de and Tsintzira, A. A. An Overview and Comparison of Technical Debt Measurement Tools. *IEEE Software* 38.03 (May 2021), pp. 61–71. ISSN: 1937-4194. DOI: 10.1109/MS.2020.3024958. URL: <https://doi.ieeecomputersociety.org/10.1109/MS.2020.3024958>.
- [36] Silva, J. D. S. da, Neto, J. G., Kulesza, U., Freitas, G., Reboucas, R. and Coelho, R. Exploring technical debt tools: A systematic mapping study. *International Conference on Enterprise Information Systems*. Springer. 2021, pp. 280–303.
- [37] Rios, N., Mendonça Neto, M. G. de and Spínola, R. O. A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* 102 (2018), pp. 117–145.
- [38] Technical Debt Tools: a Survey and an Empirical Evaluation. 12 (Aug. 2024), 8:1–8:16. DOI: 10.5753/jserd.2024.3591. URL: <https://journals-sol.sbc.org.br/index.php/jserd/article/view/3591>.
- [39] Avgeriou, P., Kruchten, P., Nord, R. L., Özkaya, İ. and Seaman, C. Reducing Friction in Software Development. *IEEE Computer Society* 33.1 (Jan. 2016), pp. 66–73. DOI: 10.1109/ms.2016.13. URL: <https://doi.org/10.1109/ms.2016.13>.
- [40] Lenarduzzi, V., Pecorelli, F., Saarimaki, N., Lujan, S. and Palomba, F. A critical comparison on six static analysis tools: Detection, agreement, and precision. *Journal of Systems and Software* 198 (2023), p. 111575. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2022.111575>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121222002515>.
- [41] Stefanović, D., Nikolić, D., Dakić, D., Spasojević, I. and Ristić, S. Static code analysis tools: A systematic literature review. *Ann. DAAAM Proc. Int. DAAAM Symp.* Vol. 31. 1. 2020, pp. 565–573.
- [42] Kazman, R. Tool support for architecture analysis and design. *Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints' 96) on SIGSOFT'96 workshops*. 1996, pp. 94–97.
- [43] Fonseca Lage, L. C. da, Kalinowski, M., Trevisan, D. and Spinola, R. Usability technical debt in software projects: A multi-case study. *2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE. 2019, pp. 1–6.
- [44] Lefever, J., Cai, Y., Cervantes, H., Kazman, R. and Fang, H. On the Lack of Consensus Among Technical Debt Detection Tools. *CoRR* abs/2103.04506 (2021). arXiv: 2103.04506. URL: <https://arxiv.org/abs/2103.04506>.

- [45] SonarSource. *SonarQube - Continuous Inspection*. Accessed: 2024-08-16. 2024. URL: <https://www.sonarqube.org>.
- [46] Developers, P. *PMD - Source Code Analyzer*. Accessed: 2024-08-16. 2024. URL: <https://pmd.github.io>.
- [47] OpenMRS. *OpenMRS Core*. <https://github.com/openmrs/openmrs-core>. Accessed: 2024-12-15. 2024.
- [48] Sharma, T., Efstathiou, V., and Spinellis, D. *On the Feasibility of Transfer-learning Code Smells using Deep Learning*. Jan. 2019. DOI: 10.48550/arxiv.1904.03031. URL: <https://arxiv.org/abs/1904.03031>.
- [49] AbuHassan, A., Alshayeb, M. and Ghouti, L. Software smell detection techniques: A systematic literature review. *Wiley* 33.3 (Oct. 2020). DOI: 10.1002/smr.2320. URL: <https://doi.org/10.1002/smr.2320>.
- [50] Silva, V. M. d. Technologies to support the technical debt management in software projects: a qualitative research. (2018).
- [51] Pollock, A. and Berge, E. How to do a systematic review. *International Journal of Stroke* 13.2 (2018), pp. 138–156.
- [52] Biolchini, J., Mian, P. G., Natali, A. C. C. and Travassos, G. H. Systematic review in software engineering. *System engineering and computer science department COPPE/UFRJ, Technical Report ES 679.05* (2005), p. 45.
- [53] Codabux, Z. and Williams, B. Managing technical debt: An industrial case study. *2013 4th International Workshop on Managing Technical Debt (MTD)*. IEEE. 2013, pp. 8–15.
- [54] Yli-Huumo, J., Maglyas, A. and Smolander, K. How do software development teams manage technical debt? – An empirical study. *Journal of Systems and Software* 120 (2016), pp. 195–218. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2016.05.018>. URL: <https://www.sciencedirect.com/science/article/pii/S016412121630053X>.
- [55] Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L. and Gorton, I. Measure it? Manage it? Ignore it? software practitioners and technical debt. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2015. Bergamo, Italy: Association for Computing Machinery, 2015, pp. 50–60. ISBN: 9781450336758. DOI: 10.1145/2786805.2786848. URL: <https://doi.org/10.1145/2786805.2786848>.
- [56] Guererro, J. *Success Stories in Managing and Paying Down Tech Debt*. Accessed: 2024-11-04. Mar. 2023. URL: <https://www.linkedin.com/pulse/success-stories-managing-paying-down-tech-debt-ioet/>.
- [57] 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 (2018), pp. 42–61. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scipro.2018.05.001>.

- org/10.1016/j.scico.2018.03.007. URL: <https://www.sciencedirect.com/science/article/pii/S0167642318301035>.
- [58] Codabux, Z. and Williams, B. Managing technical debt: An industrial case study. *2013 4th International Workshop on Managing Technical Debt (MTD)*. 2013, pp. 8–15. DOI: 10.1109/MTD.2013.6608672.
- [59] Dos Santos, P. S., Varella, A., Dantas, C. and Borges, D. Visualizing and Managing Technical Debt in Agile Development: An Experience Report. June 2013, p. 14. ISBN: 978-3-642-38313-7. DOI: 10.1007/978-3-642-38314-4_9.
- [60] Oliveira, F., Goldman, A. and Santos, V. Managing Technical Debt in Software Projects Using Scrum: An Action Research. *2015 Agile Conference*. 2015, pp. 50–59. DOI: 10.1109/Agile.2015.7.
- [61] Yin, R. K. Case Study Research: Design and Methods. *Applied Social Research Methods Series 5* (2007). An essential reference on case study research methodology.
- [62] Ampatzoglou, A., Ampatzoglou, A., Chatzigeorgiou, A. and Avgeriou, P. The financial aspect of managing technical debt: A systematic literature review. *Information and Software Technology* 64 (2015), pp. 52–73.
- [63] Team, C. D. *Cppcheck - A Static Analysis Tool for C/C++ Code*. Accessed: 2024-08-16. 2024. URL: <https://cppcheck.sourceforge.io/>.
- [64] Team, F. D. *FindBugs - Static Analysis Tool for Java*. Bill Pugh (project lead and primary developer), Andrey Loskutov (Eclipse plugin), Keith Lea (web cloud), David Hovemeyer (project founder), and other contributors. Accessed: 2024-08-16. 2024. URL: <https://findbugs.sourceforge.net/findbugs2.html>.
- [65] Developers, S. *Splint - Secure Programming Lint Tool*. Accessed: 2024-08-16. 2024. URL: <https://splint.org/>.
- [66] Wheeler, D. A. *Flawfinder - A Scanning Tool for C/C++ Source Code*. Accessed: 2024-08-16. 2024. URL: <https://github.com/david-a-wheeler/flawfinder/blob/master/README.md>.
- [67] Team, T. A. *Arcan: Software Architecture Analysis Tool*. <https://www.arcan.tech>. Accessed: 2024-09-21. 2023.
- [68] Tools, D. *Designite: A Software Design Quality Assessment Tool*. Accessed: 2024-09-23. 2024. URL: <https://www.designite-tools.com/>.
- [69] Lattix, I. *Lattix - Software Architecture Management and Analysis Tool*. Accessed: 2024-09-23. 2024. URL: <https://www.lattix.com/>.
- [70] GmbH, hello2morrow. *Sonargraph - Static Code Analysis and Architecture Management*. Accessed: 2024-09-23. 2024. URL: <https://www.hello2morrow.com/>.

APPENDIX A: EVALUATION OF TOOLS

Table A.1. Static analysis tools

Tool name	Evaluation Results
CppCheck [63]	<p>Cppcheck is an open-source static code analysis tool specifically designed for C and C++ programming languages. It focuses on identifying unsafe coding constructions and undefinable behavior and offers unique code analysis to find problems. A minimal number of false positives is the aim of this tool. The purpose of Cppcheck is to assess your C/C++ code, even if it includes non-standard syntax, which is frequently seen in embedded applications. There are two versions of Cppcheck available: the open-source version and the Cppcheck Premium version with more features and support.</p> <p>Pros :</p> <ul style="list-style-type: none"> • Cppcheck is open-source and free to use. • The tool focuses on producing fewer false positives, which means developers spend less time chasing non-existent bugs. • Users can configure the tool to suit their specific project needs, including creating custom checks. • Cppcheck can be run on multiple operating systems, including Windows, Linux, and macOS. <p>Cons:</p> <ul style="list-style-type: none"> • Cppcheck is specifically designed for C and C++ code, so it is not useful for projects written in other programming languages. <p>Pricing:</p> <ul style="list-style-type: none"> • The free version of the tool can be accessed from this website Cppcheck Official Website • The premium version which is a paid version with extended functionality is available at this page. More pricing details are available at this site.

Tool name	Evaluation Results
FindBugs [64]	<p>FindBugs is an open-source static code analysis tool designed to detect bugs in Java programs. It operates by analyzing Java bytecode, which allows it to identify potential errors that might not be visible through traditional compilation. FindBugs can detect a wide range of issues, including null pointer dereferences, infinite recursive loops, and security vulnerabilities. It helps developers improve code quality by catching bugs early in the development process.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Tailored specifically for Java, making it highly effective at finding bugs and potential issues in Java code. • The tool aims for less false positives. • The tool is equipped to find violations of recommended and essential coding practices. • Detects a wide variety of common programming errors, such as potential null pointer exceptions, deadlocks, and redundant code. <p>Cons:</p> <ul style="list-style-type: none"> • Active development of FindBugs has ceased, and it has been succeeded by other tools like SpotBugs, which continues to build on its foundation. • Difficulty in finding a proper solution when an issue arises during its configuration <p>Pricing:</p> <ul style="list-style-type: none"> • FindBugs is freely distributable under the terms and conditions <u>Lesser GNU Public License</u>

Tool name	Evaluation Results
Splint [65]	<p>Splint (short for Secure Programming Lint) is an open-source static code analysis tool designed for C programs. It extends the functionality of the traditional lint tool by adding checks specifically geared towards identifying security vulnerabilities and coding errors in C code. Splint analyzes source code to detect various potential issues, such as buffer overflows, null pointer dereferences, memory leaks, and more. It is particularly useful for developers who want to ensure their C code is robust, secure, and free from common programming mistakes.</p> <p>Pros:</p> <ul style="list-style-type: none">• Splint is particularly strong in identifying security vulnerabilities, making it an excellent tool for developers concerned with writing secure C code.• Developers can add annotations to their code to guide Splint's analysis, allowing for more precise and context-aware error detection. <p>Cons:</p> <ul style="list-style-type: none">• Splint is only applicable to C programs, so it's not useful for projects involving other programming languages.• Development of Splint has slowed down, and it is considered somewhat outdated compared to more modern tools. It may not be able to handle all aspects of contemporary C programming. <p>Pricing:</p> <ul style="list-style-type: none">• Splint is available under an open-source license, meaning it can be used, modified, and distributed at no cost.

Tool name	Evaluation Results
SonarQube[45]	<p>SonarQube is a popular open-source platform for continuous inspection of code quality. It supports a wide range of programming languages, including Java, C#, JavaScript, Python, and many others. SonarQube performs static code analysis to detect bugs, code smells, and security vulnerabilities. It integrates seamlessly with continuous integration and continuous delivery (CI/CD) pipelines, making it a valuable tool for maintaining code quality in large, complex projects. SonarQube provides comprehensive dashboards and metrics that help teams monitor and improve their codebase over time.</p> <p>Pros:</p> <ul style="list-style-type: none"> • It can be self-hosted or deployed to the cloud • Supports over 30+ programming languages, including Java, Ruby, and C • Offers integrations with popular DevOps platforms • Performs continuous code inspections • The system has the ability to classify each infraction according to its severity, ranging from minor to significant, and also provides an estimate of the required time to address the issue • Users can create “quality gates” to control that new code must exceed this gate value <p>Cons:</p> <ul style="list-style-type: none"> • May produce false positives • Free version has limited functionality <p>OS:</p> <ul style="list-style-type: none"> • Docker over Windows • macOS • Linux • Azure <p>Pricing: SonarQube is priced per instance per year and based on your lines of code. The price starts:</p> <ul style="list-style-type: none"> • For developer \$150/year/100k LOC (Lines of Code) • For developer \$20000/year/1M LOC • 14-days trial period

Tool name	Evaluation Results
PMD[46]	<p>PMD (Programming Mistake Detector) is an open-source static code analysis tool used primarily in Java, though it also supports several other languages like JavaScript, XML, Apex, and more. It helps developers identify potential issues in their codebase, such as bugs, inefficient code, or code that does not follow best practices. PMD analyzes the source code without executing it and flags common programming flaws like unused variables, empty catch blocks, unnecessary object creation, and complex expressions.</p> <p>PMD is highly extensible, allowing users to create custom rules that align with their specific coding standards. It also includes CPD (Copy-Paste Detector), which is a sub-tool that detects duplicate code across a codebase, helping to reduce redundancy and improve maintainability.</p> <p>Pros:</p> <ul style="list-style-type: none"> • PMD is free to use and modify, making it accessible to individual developers and organizations of all sizes. • While it is primarily used for Java, PMD also supports several other programming languages, making it versatile for polyglot projects. • Users can create custom rules that fit their specific coding guidelines or project needs, increasing the tool's flexibility. • PMD integrates smoothly with popular build tools like Maven, Gradle, and Ant, and can also be integrated into CI/CD pipelines, ensuring code is checked regularly. • Being an open-source tool, it has an active community that contributes to its development, offers support, and creates additional rules. <p>Cons:</p> <ul style="list-style-type: none"> • While PMD supports several languages, its primary focus is on Java. Support for other languages might not be as robust. • PMD only performs static analysis, meaning it does not analyze the code during execution, missing runtime-specific issues like concurrency problems or memory leaks. <p>Licensing:</p> <ul style="list-style-type: none"> • PMD is licensed under the BSD (Berkeley Software Distribution) License. This permissive open-source license allows you to freely use, modify, and distribute the software with minimal restrictions. You can use PMD in both personal and commercial projects without needing to pay for a license.

Tool name	Evaluation Results
Flawfinder[66]	<p>Flawfinder is an open-source static code analysis tool designed specifically for detecting security vulnerabilities in C and C++ source code. Created by David A. Wheeler, it scans code for potential flaws that attackers, such as buffer overflows, format string vulnerabilities, and race conditions might exploit. Flawfinder works by searching through the codebase and flagging common programming patterns and functions known to be risky. It then ranks these issues based on their severity, helping developers prioritize which vulnerabilities to address first.</p> <p>Flawfinder is simple to use, making it a popular choice for both security professionals and developers who want to quickly assess the security of their C/C++ projects</p> <p>Pros:</p> <ul style="list-style-type: none"> • Flawfinder specifically targets security vulnerabilities in C and C++ code, providing valuable insights for developers concerned with code safety. • The tool is command-line based and easy to set up and run, making it accessible even for those with limited experience in static analysis. • Flawfinder ranks identified issues based on their potential risk, helping developers quickly identify the most critical vulnerabilities. • Flawfinder performs quick scans of the codebase, making it a time-efficient tool for regular use during the development process. <p>Cons:</p> <ul style="list-style-type: none"> • The output of Flawfinder is text-based and somewhat basic compared to more sophisticated security analysis tools that offer graphical interfaces and detailed reports. <p>Licensing:</p> <ul style="list-style-type: none"> • Flawfinder is licensed under the GNU General Public License (GPL), which means you are free to use, modify, and distribute the software. However, if you distribute modified versions of Flawfinder, you must also release your modifications under the GPL. <p>Official website:</p> <ul style="list-style-type: none"> • The primary source for Flawfinder is its GitHub repository. This repository contains the source code, installation instructions, documentation, and other resources related to the tool.

Table A.2. Architectural and Design analysis tools

Tool name	Evaluation Results
Arcan [67]	<p>Arcan is a software analysis tool designed for code quality, architecture management, and technical debt management. It focuses primarily on the analysis of software architecture and detects architecture violations and architectural smells that may indicate poor code structure or maintainability issues. Arcan helps developers and architects to maintain clean, well-structured codebases by automatically detecting architectural drifts and violations during the development process.</p> <p>Pros :</p> <ul style="list-style-type: none"> • Arcan measures technical debt accumulated from architectural issues, giving a clear view of system maintainability over time. • The tool provides software quality analysis and (Architectural) Technical Debt detection, evaluation, and visualization. • Provides support for 5 languages: Java, C, C++, C# and Python • Works with CI/CD pipelines to ensure ongoing architecture conformance. <p>Cons:</p> <ul style="list-style-type: none"> • Not an open source software. Only paid version available. • Some users report that the documentation and resources are not as comprehensive as other tools, making it challenging for new users. <p>Pricing:</p> <ul style="list-style-type: none"> • Arcan uses a pricing model based on lines of code (LOC). The cost depends on the size of your project, so the larger the codebase, the higher the price. • Arcan typically offers 14 days trial period, which allows developers to test the tool before committing to a subscription.

Tool name	Evaluation Results
Designite[68]	<p>Designite is a software quality assessment tool focused on detecting design, architecture, and implementation smells in codebases, primarily for C# and Java. It helps developers manage technical debt by identifying problematic areas that impact maintainability and software quality.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Supports both design and implementation smells, allowing developers to tackle a wide range of issues. • Visualization tools make it easy to identify problem areas in complex codebases. • Metric customization allows tailoring the analysis to suit project-specific requirements. <p>Cons:</p> <ul style="list-style-type: none"> • Limited language support, focusing only on C# and Java, though this is sufficient for many teams. <p>Pricing:</p> <ul style="list-style-type: none"> • Designite offers a trial version with limited features that can be used indefinitely. For full access to all features, a professional license is required. • Professional C# license is 400\$ per year and Professional Java license is 250\$ per year.

Tool name	Evaluation Results
Lattix[69]	<p>Lattix is a popular software architecture management and analysis tool designed to help organizations understand, manage, and reduce technical debt. Lattix provides tools to analyze, visualize, and refactor software architecture to minimize architectural issues. Lattix creates detailed visual models of the system's architecture, which include dependencies between components. This helps developers and architects better understand the structure of complex software systems.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Supports various programming languages such as Java, C/C++, C#, Python • Continually tracks architectural violations and debt metrics, giving teams up-to-date insights into code quality. <p>Cons:</p> <ul style="list-style-type: none"> • Lattix is a commercial tool, and its pricing may be high for smaller organizations or startups with limited budgets. <p>Pricing:</p> <ul style="list-style-type: none"> • Lattix typically offers a free trial period, usually 14 or 30 days. • Pricing depends on the number of users, the scale of the project, and the features required.

Tool name	Evaluation Results
Sonargraph[70]	<p>Sonargraph is a software architecture and quality management tool designed to help developers and architects manage technical debt, detect architectural violations, and improve code quality. It provides features such as architecture visualization, which offers a graphical representation of the system's structure and dependencies, along with static code analysis to detect code smells and design flaws. Sonargraph supports rule-based architecture compliance, ensuring that code adheres to predefined design guidelines, and offers hotspot detection to identify problematic areas in the codebase.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Supports multiple programming languages including Java, Kotlin (the JVM version of it), C#, Python 3, TypeScript and C/C++ • Seamless integration with CI/CD tools ensures ongoing monitoring and alerts when code quality or architectural standards are violated. • Focuses on maintaining and monitoring architectural compliance, which is particularly useful for large, complex projects with strict design guidelines. <p>Cons:</p> <ul style="list-style-type: none"> • While Sonargraph offers incremental analysis, the initial scan for large, complex projects may still require significant time and computational resources. <p>Editions:</p> <ul style="list-style-type: none"> • Sonargraph Architect: Focused on architecture modeling and monitoring. • Sonargraph Developer: The Sonargraph-Developer license allows developers to check for issues either by using our plugins for Eclipse or IntelliJ and/or by using the Sonargraph-Architect application in read-only mode in parallel to your IDE. • Sonargraph-Enterprise: Sonargraph-Enterprise, consolidates metrics for all projects within an organization in one place. This allows users to answer key questions such as: Which projects have experienced the largest size increase over the past 30 days? Which projects have the lowest maintainability? And which projects have undergone the most significant relative changes in maintainability? <p>Pricing:</p> <ul style="list-style-type: none"> • Sonargraph is offered under a commercial license, typically priced based on factors such as programming language, product edition, and team size. • Sonargraph offers both annual subscriptions and perpetual licenses, depending on your needs. • Free trial is available

Tool name	Evaluation Results
JSPIRIT	<p>JSPIRIT is a versatile tool that allows developers to add custom code smell detection strategies and prioritize them based on specific criteria. It can also detect clusters, or "agglomerations," of interrelated code smells that may indicate architectural issues. JSPIRIT currently supports two types of agglomerations: within a component, where related code smells are grouped within the same component, and within a hierarchy, where the same code smell appears across an inheritance tree, suggesting a broader structural problem.</p> <p>Pros:</p> <ul style="list-style-type: none"> • JSPIRIT supports the identification of 10 code smells such as God class, Feature envy, intensive coupling etc • JSPIRIT offers detailed insights into code quality metrics, making it easier for developers to track the maintainability of the project over time and prioritize areas for improvement. • The tool provides automated refactoring suggestions, which can save time for developers. <p>Cons:</p> <ul style="list-style-type: none"> • JSPIRIT is specific to Java, so it may not be suitable for organizations with projects in multiple programming languages. <p>Pricing :</p> <ul style="list-style-type: none"> • Free version available

Table A.3. Test Management Tools

Tool name	Evaluation Results
TestLodge	<p>TestLodge is a cloud-based test management tool designed to streamline the software testing process and manage test debt. It provides teams with a comprehensive platform to create, manage, and execute test plans and test cases while enabling effective collaboration and communication among team members. TestLodge is particularly beneficial for QA teams looking to enhance their testing workflows, maintain clear documentation, and integrate with various development and testing tools to ensure high-quality software delivery.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Users can create detailed test plans that outline the scope and objectives of testing activities. This feature helps ensure that all necessary tests are covered and aligned with project requirements. • It can build a suite of test cases using a simple interface, or the tester can import existing test cases from a spreadsheet. • The tool guides through each test, where the test can be marked as passed, failed, or skipped. • The tool is customizable <p>Cons:</p> <ul style="list-style-type: none"> • Pricing • Learning curve <p>Pricing</p> <ul style="list-style-type: none"> • Basic model starts from \$69 per month • Free trial available.

Tool name	Evaluation Results
TestSigma	<p>Testsigma takes into account elements like the simplicity of test creation, scalability, and integration options. As a robust test automation platform, Testsigma effectively helps manage test debt. Its AI-driven test maintenance and codeless automation features alleviate the challenges associated with maintaining test scripts.</p> <p>Pros:</p> <ul style="list-style-type: none"> • Testsigma uses AI to automatically update tests when the application changes, reducing test debt. • Testsigma can integrate with your CI/CD pipeline and other testing tools for a streamlined testing process. • Tool can be accepted for both manual and automation testing because it will help handle frequent changes, faster execution, and feedback. • Testsigma helps minimize test debt by offering codeless automation and AI-driven test maintenance. <p>Cons:</p> <ul style="list-style-type: none"> • Testing agent issues reported by users in community chat. • Not an open-source software. <p>Pricing:</p> <ul style="list-style-type: none"> • Free trial available • Enterprise and pro model pricing depends on team size and type of application under test.