UNDERSTANDING ASSET DEGRADATION IN SOFTWARE ENGINEERING

Ehsan Zabardast

Blekinge Institute of Technology
Doctoral Dissertation Series No. 2023:05

Department of Software Engineering

Understanding Asset Degradation in Software Engineering

Ehsan Zabardast

Blekinge Institute of Technology Doctoral Dissertation Series
No 2023:05

Understanding Asset Degradation in Software Engineering

Ehsan Zabardast

Doctoral Dissertation in Software Engineering

Department of Software Engineering
Blekinge Institute of Technology
SWEDEN

Department of Software Engineering
Publisher: Blekinge Institute of Technology
SE-371 79 Karlskrona, Sweden
Printed by Media-Tryck, Lund, Sweden, 2023
ISBN: 978-91-7295-455-7
ISSN: 1653-2090
urn:nbn:se:bth-24429

To Nikoo.

"To be confused is really a very exciting thing."
"Science is the desire to remain confused."
- Robin Ince & Brian Cox; The Infinite Monkey Cage

Abstract

Background: As software is everywhere, and almost every company has nowadays a dependency on software, designing and developing software-intensive products or services has become significantly challenging and time-consuming. The challenges are due to the continuous growth of the size and complexity of software and the fast pace of change. It is important that software-developing organisations' engineering practises adapt to the rising challenges by adopting well-engineered development activities.

Organisations deal with many software artefacts, some of which are more relevant for the organisation. We define Software Assets as artefacts intended to be used more than once. Given software development's continuous and evolutionary aspect, the assets involved degrade over time. Organisations need to understand what assets are relevant and how they degrade to exercise quality control over software assets. Asset degradation is inevitable, and it may manifest in different ways.

Objective: The main objective of this thesis is: (i) to contribute to the software engineering body of knowledge by providing an understanding of what assets are and how they degrade; and (ii) to gather empirical evidence regarding asset degradation and different factors that might impact it on industrial settings.

Method: To achieve the thesis goals, several studies have been conducted. The collected data is from peer-reviewed literature and collaboration with five companies that included extracting archival data from over 20 million LOC and archival data from open-source repositories.

Results: The first contribution of this thesis is defining the concept of assets and asset degradation in a position paper. We aim to provide an understanding of software assets and asset degradation and its impact on software development.

Additionally, a taxonomy of assets is created using academic and industrial input. The taxonomy includes 57 assets and their categories.

To further investigate the concept of asset degradation, we have conducted in-depth analyses of multiple industrial case studies on selected assets. This thesis presents results to provide evidence on the impact of different factors on asset degradation, including: (i) how the accumulation of technical debt is affected by different development activities; (ii) how degradation 'survives'; and (iii) how working from home or the misalignment between ownership and contribution impacts the faster accumulation of asset degradation. Additionally, we created a model to calculate the degree of the alignment between ownership and contribution to code.

Conclusion: The results can help organisations identify and understand the relevant software assets and characterise their quality degradation. Understanding how assets degrade and which factors might impact their faster accumulation is the first step to conducting sufficient and practical asset management activities. For example, by engaging (i) proactively in preventing uncontrolled growth of degradation (e.g., aligning ownership and contribution); and (ii) reactively in prioritising mitigation strategies and activities (focusing on recently introducing TD items).

Keywords: *Assets in Software Engineering, Asset Management, Asset Degradation, Technical Debt*

Acknowledgements

I am honored to have this opportunity to express my gratitude to those who have contributed to my doctoral journey. Without the support, encouragement, and guidance of many individuals, this dissertation would not have been possible.

I would like to start by expressing my sincere appreciation to my supervisors, Dr. Javier Gonzalez Huerta, Prof. Dr. Darja Šmite, and Prof. Dr. Tony Gorschek, who have been excellent mentors throughout this journey. Their expertise, patience, and continuous feedback have been invaluable in shaping my research and ensuring that I remained focused and motivated.

I would like to thank all my colleagues at SERL Sweden and BTH, who are among the best people I know. I am grateful for all their friendship, feedback, support, and unwavering love. In particular, I would like to thank Daniel Mendez, Davide Fucci, Michael Dorner, Julian Frattini, Andreas Bauer, Felix Jedrzejewski, Svetlana Zivanovic, Anna Eriksson, Anders Sundelin, Oleksandr Kosenkov, Parisa Elahidoost, Lukas Thode, Krzysztof Wnuk, Niklas Lavesson, and Waleed Abdeen not only for all their help and support but also for all the fun and relaxation moments.

I would like to extend my heartfelt gratitude to my wife, Nikoo, who has been my pillar of strength throughout my doctoral journey. Her unwavering support, patience, and encouragement have sustained me through the highs and lows of this challenging process. I am grateful for her sacrifices and understanding during the times when I had to spend long hours working on my research. I would also like to express my appreciation to my parents, Hossein and Roghaiyeh, who have always been my inspiration and source of motivation. Their unconditional love and support have been the driving force behind my academic pursuits. I am grateful for their unwavering belief in my abilities and their encouragement to pursue my dreams.

Overview of Publications

Publications included in this thesis

This compilation thesis includes the following seven paper.

- **Chapter 2: Ehsan Zabardast**, Julian Frattini, Javier Gonzalez-Huerta, Daniel Mendez, Tony Gorschek, and Krzysztof Wnuk. "Assets in Software Engineering: What are they after all?". *In Journal of Systems and Software*, (pp. 111485), Elsevier (2022).
  doi: `https://doi.org/10.1016/j.jss.2022.111485`
- **Chapter 3: Ehsan Zabardast**, Javier Gonzalez-Huerta, Tony Gorschek, Darja Šmite, Emil Alégroth, and Fabian Fagerholm. "A Taxonomy of Assets for the Development of Software-Intensive Products and Services". *In Journal of Software and Systems*, (pp. 111701), Elsevier (2023).
  doi: `https://doi.org/10.1016/j.jss.2023.111701`
- **Chapter 4: Ehsan Zabardast**, Javier Gonzalez-Huerta, and Darja Šmite. "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study". *In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, (pp. 376-384), IEEE (2020).
  doi: `https://doi.org/10.1109/SEAA51224.2020.00068`
- **Chapter 5: Ehsan Zabardast**, Kwabena Ebo Bennin, and Javier Gonzalez-Huerta. "Further investigation of the survivability of code technical debt items". *In Journal of Software: Evolution and Process*, 34(2):e2425, Wiley (2022).
  doi: `https://doi.org/10.1002/smr.2425`

- **Chapter 6: Ehsan Zabardast**, Javier Gonzalez-Huerta, and Binish Tanveer. "Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution". *In 2022 19th International Conference on Software Architecture Companion (ICSA-C)*, (pp. 30-34), IEEE (2022).

  doi: `https://doi.org/10.1109/ICSA-C54293.2022.00013`

- **Chapter 7: Ehsan Zabardast**, Javier Gonzalez-Huerta, and Francis Palma. "The Impact of Forced Working-From-Home on Code Technical Debt: An Industrial Case Study". *In 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, (pp. 298-305), IEEE (2022).

  doi: `https://doi.org/10.1109/SEAA56994.2022.00054`

- **Chapter 8: Ehsan Zabardast**, Javier Gonzalez-Huerta, Francis Palma, and Panagiota Chatzipetrou. "The Impact of Ownership and Contribution Alignment on Code Technical Debt Accumulation". *Submitted to Transactions on Software Engineering*, (Under Review), IEEE (2023).

  arXiv preprint: `https://doi.org/10.48550/arXiv.2304.02140`

Contribution Statement

Ehsan Zabardast is the first author for all the papers that are included in this thesis. As the leading author, Ehsan was responsible for designing and conducting the studies, collecting and analysing data, and most reporting activities. In Addition, he is the sole author of Chapter 1, the Introduction. The detailed authors contribution to the chapters are described below using the contributor role taxonomy [112]:

Chapter 2

- *Ehsan Zabardast:* Conceptualisation, Visualisation, Writing - original draft, Writing - review & editing
- *Julian Frattini:* Conceptualisation, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Conceptualisation, Funding acquisition, Writing - original draft, Writing - review & editing

- *Daniel Mendez:* Conceptualisation, Writing - original draft, Writing - review & editing
- *Tony Gorschek:* Conceptualisation, Funding acquisition, Writing - review & editing
- *Krzysztof Wnuk:* Conceptualisation, Writing - review & editing

Chapter 3

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Validation, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Conceptualisation, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Project administration, Supervision, Validation, Writing - original draft, Writing - review & editing
- *Tony Gorschek:* Conceptualisation, Data curation, Formal analysis, Funding acquisition, Project administration, Supervision, Writing - review & editing
- *Darja Šmite:* Formal analysis, Data curation, Supervision, Writing - review & editing
- *Emil Alégroth:* Data curation, Formal analysis, Investigation, Writing - review & editing
- *Fabian Fagerholm:* Data curation, Investigation, Writing - review & editing

Chapter 4

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Conceptualisation, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Supervision, Writing - review & editing

- *Darja Šmite:* Methodology, Supervision, Writing - review & editing

Chapter 5

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Visualisation, Writing - original draft, Writing - review & editing
- *Kwabena Ebo Bennin:* Investigation, Methodology, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Funding acquisition, Data curation, Investigation, Methodology, Writing - review & editing

Chapter 6

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Validation, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Formal analysis, Funding acquisition, Investigation, Methodology, Validation, Writing - original draft, Writing - review & editing
- *Binish Tanveer:* Investigation, Methodology, Validation, Writing - original draft, Writing - review & editing

Chapter 7

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Conceptualisation, Data curation, Formal analysis, Investigation, Funding acquisition, Methodology, Visualisation, Writing - original draft, Writing - review & editing
- *Francis Palma:* Methodology, Visualisation, Writing - original draft, Writing - review & editing

Chapter 8

- *Ehsan Zabardast:* Conceptualisation, Data curation, Formal analysis, Investigation, Methodology, Project administration, Validation, Visualisation, Writing - original draft, Writing - review & editing
- *Javier Gonzalez-Huerta:* Conceptualisation, Data curation, Formal analysis, Funding acquisition, Investigation, Methodology, Validation, Visualisation, Writing - original draft, Writing - review & editing
- *Francis Palma:* Methodology, Visualisation, Writing - original draft, Writing - review & editing
- *Panagiota Chatzipetrou:* Formal analysis, Writing - review & editing

Other publications not included in this thesis

- Michael Dorner, Maximilian Capraro, Oliver Treidler, Tom Eric Kunz, Darja Šmite, **Ehsan Zabardast**, Daniel Mendez, and Krzysztof Wnuk. "Taxing Collaborative Software Engineering". *Sumbitted to IEEE Software*, (Under Review), IEEE (2023).
- Binish Tanveer, **Ehsan Zabardast**, and Javier Gonzalez-Huerta. "An approach to align socio-technical dependencies in large-scale software development". *In 2023 20th International Conference on Software Architecture Companion (ICSA-C)*, (Accepted), IEEE (2023).
- Davide Fucci and **Ehsan Zabardast**. "Towards Group Development Stages in Software Engineering Courses Projectwork". *In Journal of Teaching and Learning in Higher Education at Malmö University*, 1(2), Malmö University (2020).

  doi: `https://doi.org/10.24834/jotl.1.2.587`
- Aivars Šāblis, Javier Gonzalez Huerta, **Ehsan Zabardast**, and Darja Šmite. "Building LEGO towers: an exercise for teaching the challenges of global work". *In ACM Transactions on Computing Education (TOCE)*, 19(2), 1-32, 2019.

  doi: `https://doi.org/10.1145/3218249`

Funding

This research was supported by the KK foundation through the SHADE KK-Hög project under grant 2017/0176 and Research Profile project SERT under grant 2018/010 at Blekinge Institute of Technology, SERL Sweden.

Contents

List of Abbreviations

Abbreviation	Definition
AD	Asset Degradation
ADA	Author Defined Assets
AM	Asset Management
API	Application Program Interface
DRM	Design Research Methodology
OCAM	Ownership and Contribution Alignment Model
OSS	Open Source Software
QA	Quality Attribute
RUP	Rational Unified Process
SDP	Software Development Process
SE	Software Engineering
SIPS	Software-Intensive Products and Services
SLR	Systematic Literature Review
SME	Small and Medium-sized Enterprise
SPL	Software Product Lines
SWEBOK	Software Engineering Body of Knowledge
TD	Technical Debt
$TD_{Density}$	Technical Debt Density
TDD	Technical Debt Density
TDI	Technical Debt Item
TDM	Technical Debt Management

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Overview

Designing, developing, and maintaining software-intensive products and services is a complex and significant undertaking with many challenges. There are a myriad of decisions that need to be taken by stakeholders during all stages of software development [106]. The fast pace of software development may lead to sub-optimal decisions that impact the maintenance and evolvability of the software [14].

There are many software artefacts that are created, maintained, and evolved during software development and are essential to the final product or service's quality. The quality of software artefacts that are exposed to continuous change and increasing complexity may decrease unless proper quality assurance is exercised on them [116]. Therefore, it is crucial that software developing organisations be able to identify software artefacts that are involved in the development process [143].

The continuous maintenance and evolution of software impacts the artefacts involved in the development process. The constant use, maintenance, and evolution causes software artefacts to degrade in quality. The degradation of software artefacts needs to be managed since they have value for stakeholders [143]. Furthermore, the degradation of some of these artefacts might have bigger impact on the organisation as they are more relevant, i.e., they are used often over time by the stakeholders in a given organisation.

There are various factors that impact the quality degradation of software artefacts. And there have been many studies investigating software artefacts' quality from different perspectives, for example [3, 64, 127, 167, 214]. Practitioners and researchers use metrics to evaluate the quality of software artefacts. One of the metrics that has become popular both in industry and in academia is technical debt [178]. Technical debt has traditionally been used to measure the consequences of short-term decisions on long-term software development [14, 106]. The technical debt metaphor has been extended and studied by many researchers [178], it has been an interesting topic for both academia and industry, and it has grown from a metaphor to a practice [107].

A lot of research has been carried out regarding specific and well-known software artefacts and how they degrade, for example [118, 128, 137, 178]. While the lack of awareness of software artefacts might not pose significant challenges in developing software-intensive products or services, identifying software artefacts that are used often and whose quality needs to be controlled is necessary for reach more effective artefact management activities. In other words, organ-

isations can invest in exercising quality control activities on artefacts that are used often and are more relevant.

Another aspect related to quality degradation is propagation of degradation among different artefacts, i.e., degradation of an artefact worsening the degradation of other related artefacts. Several studies have investigated the propagation of quality degradation among software artefacts (e.g., [2, 198]). Therefore, raising the awareness and identification of software artefacts with greater value for organisations and their quality degradation has the potential to help organisations to be more successful and efficient when developing software-intensive products or services.

> ***Software assets*** *are those relevant software artefacts that have long-term strategic value —they are intended to be used more than once— and can provide a competitive advantage to the organisation. These may include reusable software components, libraries, and other intellectual property that can be leveraged across multiple projects and products. Proper management of software assets can lead to significant cost savings, increased productivity, and faster time-to-market for new software solutions.*

This thesis summarises the findings of empirical investigations regarding the identification of software assets, that are relevant for the development organisations, and their quality degradation. The contribution and the flow of the research presented in this thesis are illustrated in Figure 1.1. The research gaps G1 and G2 are presented Section 1.3. The research approach using design research methodology (DRM) is presented in Section 1.5. And the thesis contribution are presented in Section 1.6.

The main aim of this thesis is two-fold: i) help organisations understand and identify relevant software assets; and ii) provide empirical evidence illustrating quality degradation of software assets, as well as organisation factors that might exacerbate this quality degradation.

Furthermore, this thesis intends to foster improvements by providing: a taxonomy of relevant software assets and an empirically created model to assess the alignment degree between ownership and contribution.

The rest of the this chapter is structured as follows: Section 1.2 contains relevant background together with a discussion of the related work. The research gaps are presented in Section 1.3. The addressed research questions in this thesis are presented in Section 1.4. The research approach, methodologies used in this thesis, and the threats to validity are presented in Section 1.5. The

contributions of the work is summarised in Section 1.6. Section 1.7 present the discussion followed by conclusions and future work in Section 1.9.

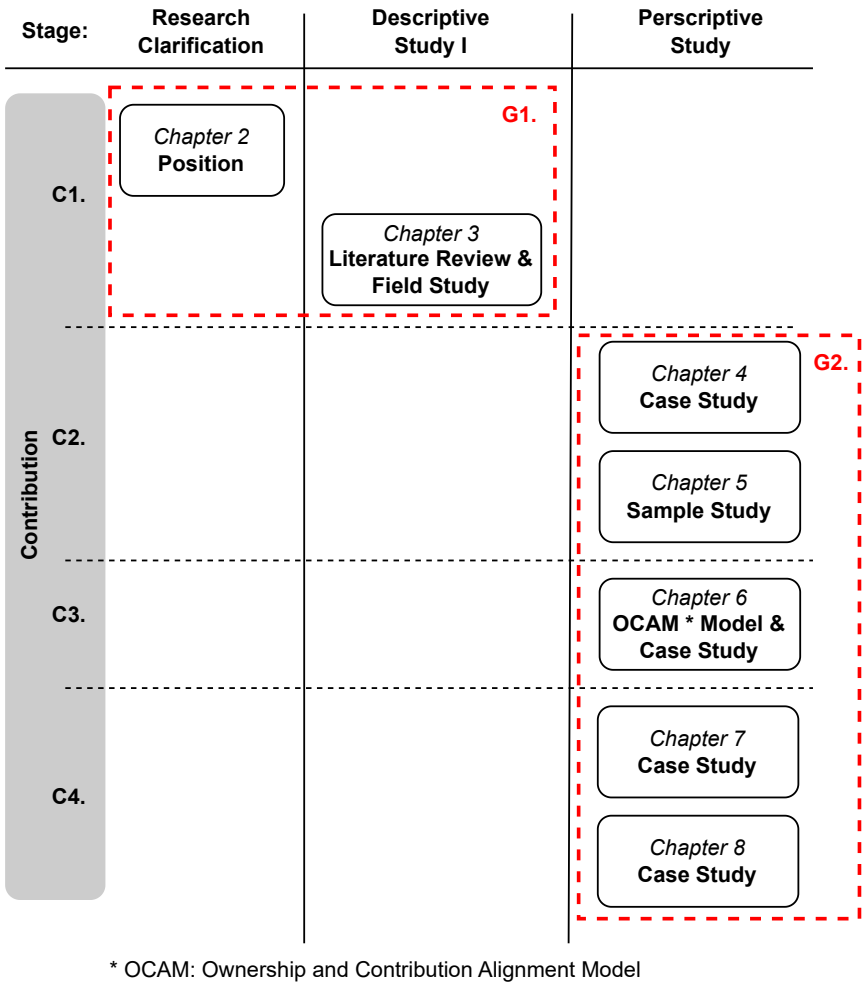

Figure 1.1: Overview of the chapters and thesis contribution. Boxes denote how chapters of this thesis map to the contributions, research gaps, and design research methodology (DRM) framework stages [33].

1.2 Background and Related Work

The section summarises the background for the thesis. The research conducted and presented in this thesis mainly relates to four topics: i) Artefacts in Software Engineering, ii) Software Evolution, and iii) Technical Debt.

1.2.1 Artefacts in Software Engineering

Software artefacts are of high importance to software development. Documents, deliverable, and work products are examples of software artefacts. They have been around since the beginning of software development practices [143]. Software artefacts have been defined in different sources. A software artefact is:

- "Documentation of the results of development steps" [45].
- "A work product that is produced, modified, or used by a sequence of tasks that have value to a role" [141].
- "A self-contained work result, having a context-specific purpose and constitutes a physical representation, a syntactic structure and a semantic content of said purpose, forming three levels of perception" [141].

Understanding software artefacts, how they are structured, and how they relate to each other has a significant influence on software-intensive developing organisations [45]. The need for organising software artefacts structurally is highlighted as the documentation in large-scale systems grow exponentially [45].

Artefacts defined by most of the Software Development Processes (SDPs) are monolithic and unstructured [204]. The content of poorly structured artefacts is difficult to reuse, and the evolution of such monolithic artefacts is cumbersome [190]. Therefore, different SDPs present various models for presenting software artefacts, e.g., the Rational Unified Process (RUP) [103, 104]. There are ways to classify and structure software artefacts based on well-known modelling concepts. Examples of such models are the work of Broy [45] and Silva et al. [190]. Moreover, there are ontologies and meta-models to classify artefacts in specific software development areas (e.g., Idowu et al. [89] Mendez et al. [142], Zhao et al. [232], and Constantopoulos and Doerr [52]).

However, the definitions of artefacts presented in the literature do not distinguish between artefacts that have an inherent value for the development organisation or artefacts that are intended to be used more than once. There is a lack of definition of such artefacts and the terminology around them. Software

developing organisations can improve quality control practises by identifying these important artefacts.

It is important to identify and distinguish relevant software artefacts, i.e., software assets, whose quality needs to be controlled since they have different characteristics and value propositions for software development organisations.

On the other hand, software artefacts are temporary or disposable items that are created during the software development process. While artefacts are essential for managing the development process and documenting the software system, they do not have the same long-term value as software assets.

By identifying and distinguishing software assets from software artefacts, organisations can develop effective strategies for managing their quality degradation. They can prioritise the development and maintenance of assets that provide the most value, while streamlining the management of artefacts to reduce waste and improve efficiency. This can lead to more effective software development practices, better quality software solutions, and improved business outcomes.

1.2.2 Software Evolution

Software evolution refers to the process of changing and adapting software systems over time in response to changing requirements, technologies, and user needs. It is an ongoing process that involves the modification, enhancement, and maintenance of existing software systems throughout their lifecycle.

Software evolution is a natural and necessary aspect of software development [177], as software systems are rarely developed once and then left untouched. Instead, they typically evolve in response to changing requirements, bug fixes, and new features. Evolution can also be driven by external factors such as changes in the underlying technology stack, security vulnerabilities, and user feedback.

Software products and services, due to their nature, evolve [177]. The development process for software-intensive products or services is continuous, i.e., software products or services constantly change to include new features and functionality or to fix bugs or provide improvements [114, 116]. And as the software product or service is changing and growing, it's complexity increases, unless work is done to maintain or reduce it [116]. Therefore, there is a need for constantly maintaining the software to keep the quality satisfactory [114, 116].

The rapid change of software, i.e., the ability to rapidly evolve software, creates challenges for software-developing organisations to develop reliable, high-quality software [145]. Software changes have been studied comprehensively

(e.g., [47, 129]). The literature categorises software changes into *corrective*, *adaptive*, *perfective*, and *preventive* modifications [177]. Such modifications are discussed in the scope of software systems and products and in relation to the degradation of the external functionality.

Software artefacts that are involved in the development of any software-intensive product evolve as new functionality and improvements are implemented in the product or service. The evolution and change of software artefacts implies that their quality will diminish if proper quality control practices are not practised [114, 116]. Therefore, the quality degradation of software artefacts due to their evolution is important to consider when planning management activities.

1.2.3 Technical Debt

Software artefacts are impacted by the evolution of software product or service, i.e., due to change and maintenance. Evolving software artefacts need to adhere to a certain level of quality or standards. When a stakeholder does not keep the quality of any artefact to its standard, they incur technical debt.

Technical debt (TD) was introduced by Cunningham in 1992 to describe the consequences of making sub-optimal decisions made to gain immanent goals [55]. The TD metaphor has been extended to include more than source-code-related artefacts, and it is currently one of the topics receiving more attention in software engineering [14, 178]. The metaphor has been studied in numerous articles in academia, and it has been incorporated as a concept widely used and discussed by industrial practitioners as it gains popularity [178]. At the moment, TD is one of the critical issues in the software development industry, and if it remains unchecked can lead to significant cost increases [30, 106, 178].

TD is recognised as one of the critical issues in the software development industry [29, 178]. TD is "pervasive", and it includes all aspects of software development, signifying its importance both in the industry and academia [106]. The activities that are performed to prevent, identify, monitor, measure, prioritise, and repay TD are called Technical Debt Management (TDM) [14, 85] and include such activities as, for example, identifying TD items in the code [212, 218], visualising the evolution of TD [63, 162], evaluating source code state [123, 212], and calculating TD principal [7, 56].

As the TD metaphor was extended to include different aspects of software development, various TD types were introduced [14], e.g., requirements debt, test debt, and documentation debt. The introduction of different types of TD has led researchers to attempt to classify the different types and categories of TD.

One of the earliest classifications of TD is the work of Tom et al. [206]. Other secondary and tertiary studies have been performed to summarise the current state of TD and TD types, e.g., by Lenarduzzi et al. [118], Rios et al. [178], and Li et al. [128].

The metaphor has been mainly *operationalised* by researchers and practitioners for source code, i.e., code, design, and architecture [14]. The organisational, environmental and social aspects of TD have not received the same amount of attention [178]. Moreover, TD types are often studied in isolation, not considering how one TD can be contagious to other assets and the permeation to other TD types [14]. For example, when the TD in code grows, a valid question would be whether and how it impacts the TD in tests and the extent of such impact.

Technical debt management activities have been investigated in several secondary studies with different perspectives, some focusing on tools, others on strategies, but there is still a lack of unified analysis aligning these different perspectives [178].

TD is generally studied in the current state of the software, and it is understudied with regards to the evolutionary aspects of TD, i.e., studying TD on a "snapshot" of a system is not enough [68, 171]. Therefore, a more appropriate approach to studying TD is to study its evolution [43].

There are articles on the implications of the impact of low ownership of a module on code quality (e.g., [170]). However, to best of our knowledge, there has not been any study on the impact of team ownership and contribution alignment on the faster accumulation of TD. Finally, there are limited studies on the impact of organisational factors, such as team structure or ways-of-working on the faster accumulation of TD.

1.3 Research Gaps

The existing literature has investigated and defined software assets. There is a need to organise and structurise software artefacts [142, 204] that are valuable for software developing organisations, i.e., software assets. Currently, there is no structured way with which an organisation can define and identify the valuable artefacts. The first gap addressed in this research is:

> ***G1.*** *Lack of a definition of software assets, asset degradation, and a taxonomy of such assets.*

Consequently, lack of a definition may lead to fuzzy, unclear, and or even contradictory understanding of the essential concepts involved such as software

assets, which is essential to further study the change of quality of assets over time, i.e., asset degradation. Providing empirical evidence, especially from industry(e.g., [2, 198, 210]), solidifies the understanding of said assets. However, there is a need of more empirical evidence supporting asset degradation [178]. Understanding different organisational factors that might have an impact on the faster degradation of assets is the next step to achieving this goal. The second gap addressed in this thesis is:

> ***G2.*** *The need to provide empirical evidence from industrial cases supporting and illustrating the concept of asset degradation, and how different organisational factors might impact the faster accumulation of assets' quality degradation.*

This thesis fills the existing gaps by employing empirical research using quantitative and qualitative analysis of data collected from industry. The next section provides the research questions address in this thesis and how they address the mentioned gaps.

1.4 Research Questions

This thesis addresses the following research questions:

> **RQ1.** What is asset degradation in software engineering?

Developing software-intensive products and service requires utilisation of many various "assets", denoting the inherent value of assets to the organisation. There are terms and concepts used in software engineering that are often intermixed and not used consistently. This is evident when examining the literature on technical debt, and it is observed while working with our industrial partners. The role of assets in software engineering, management, and evolution is paramount. Yet, there is no clear consensus as to which artefacts are assets when dealing with software-intensive products and services. Therefore, a structured terminology is required to clearly define the distinctive characteristics of assets that allows us to differentiate assets and that allows us to frame the asset management concepts, such as "asset degradation".

To answer this research question, we started with positioning the concept of assets and defining the related concepts, i.e., asset degradation and asset management (Chapter 2). We have identified the different types of asset degradation namely deliberate, unintentional, and entropy. And we have discussed

their differences with examples. We discuss the propagation of degradation on assets. Degradation propagation is the influence of assets' degradation to other dependent assets. Finally, we created a taxonomy of assets while considering both academic and industrial input. The taxonomy was created based on a literature review and a field study (Chapter 3).

This thesis explores the ideas and concepts introduced and coined during the research. The research conducted to answer is exploratory and it is conducted to find evidence and indications to support the assumptions. The quality degradation of certain software assets have been studied before as mentioned in Section 1.2. However, there is a growing interest to study the degradation of assets and its implications through case studies, sample studies, and field studies.

> **RQ2.** What factors impact the degradation of assets when developing software-intensive products or services?

After clarifying the concepts and terminology and identifying the assets, this research question deals with understanding the impact of different factors on asset degradation, starting with organisational factors. The thesis aims at understanding the organisational factors that might impact degradation of certain assets. Understanding and identifying such factors is necessary to create methods and models to effectively manage asset degradation. However, the creation of such models is outside of the scope of this thesis.

Given that by defining the concepts and creating the taxonomy, we covered a wide range of software development body of knowledge, evaluating and measuring all assets requires tremendous effort and much time. Therefore, we started by investigating a subset of assets. We decided to begin with assets related to the source code and architecture.

There are well-defined metrics for source-code- and architecture-related assets, and data can be retrieved both from industrial projects and open source systems. Moreover, starting with a smaller set of assets will make it easier to create and evaluate methods that can eventually be used for other assets.

To answer this research question, we started by examining the existing metrics and methods to evaluate source-code- and architecture-related assets. We performed studies to investigate asset degradation first to identify different factors related to the degradation of assets, and second to evaluate their impact on asset degradation using industrial data. These case studies are presented in Chapters 4, 5, 6, 7, and 8. Additionally, we created a model to measure the degree of alignment between the ownership and contribution that led to study-

ing such alignment's impact on code degradation, i.e., by faster accumulation of technical debt.

1.5 Research Approach

This section provides an overview of the research approach in this thesis. The section contains a summary of the research methods and empirical data sources used in this thesis. This section is concluded by presenting the limitations and threats to validity of the thesis.

The research presented in this thesis utilises both quantitative and qualitative research methods and data. Therefore, a mixed-methods research approach [54] is used in order to answer the research questions mentioned in the previous section.

Most of the research done in this thesis is empirical in its nature. Empirical methods, e.g., case studies, aimed at investigating scientific evidence on phenomena related to software development, operation, and maintenance [66]. Empirical research relies on the collaboration of academia and industry, where scientific evidence is created by investigating industrial cases, and it is used to support practitioners. A brief introduction to the different research methods presented in this thesis is provided next; an overview of the different methods, along with the contributions of individual studies, is depicted in Table 1.1.

The research presented in this thesis uses the design research methodology (DRM) framework as a guideline to systematically design the research [33]. The DRM framework is proactively used to structurise the content of this thesis.

The DRM framework consist of four stages namely *Research Clarification*, *Descriptive Study I*, *Prescriptive Study*, and *Descriptive Study II* [33]. Figure 1.2 illustrates the stages, the link between stages, basic means, the main outcome from each stage, and the mapping of the chapters presented in this thesis to the DRM framework. The bold arrows between the stages illustrate the main process flow, the light arrows the many iterations. The *research clarification* stage is where the researchers find evidence or indications to support their assumptions. The goal of the second stage, i.e., *descriptive study I*, is to help researchers determine which factors to address and obtain a better understanding of the phenomena before moving to the next stage. In the *prescriptive study*, researchers use their previously found understanding about the phenomena to correct and elaborate on their initial description in the first stage. And finally, in the *descriptive study II* stage, the researchers investigate the impact

of the support and its ability to realise the desired situation. There are seven main types of design research within the DRM framework.

The research presented in this thesis follows the second type of design research, i.e., *Comprehensive Study of Existing Situation* which contains the first three stages of the DRM framework. The rest of this section presents an overview of the stages.

1.5.1 Stage 1: Research Clarification

This stage is dedicated to find indications or evidence that supports the assumptions that lead to the research goal. During this stage, the researchers provide an initial description of the existing and the desired situation. Chapter 2 presents the Research Clarification stage in this thesis.

Table 1.1: Overview of the research approach, data collection methods, and analysis methods presented in this thesis.

	Chapters						
Research Approach	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Literature Review		•					
Field Study		•					
Case Study			•		•	•	•
Sample Study				•			
Data Collection Method	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Snowballing		•					
Focus Group		•			•	•	•
Archival Data Retrieval			•	•	•	•	•
Analysis Method	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Statistical Analysis			•	•	•	•	•
Thematic Analysis		•				•	

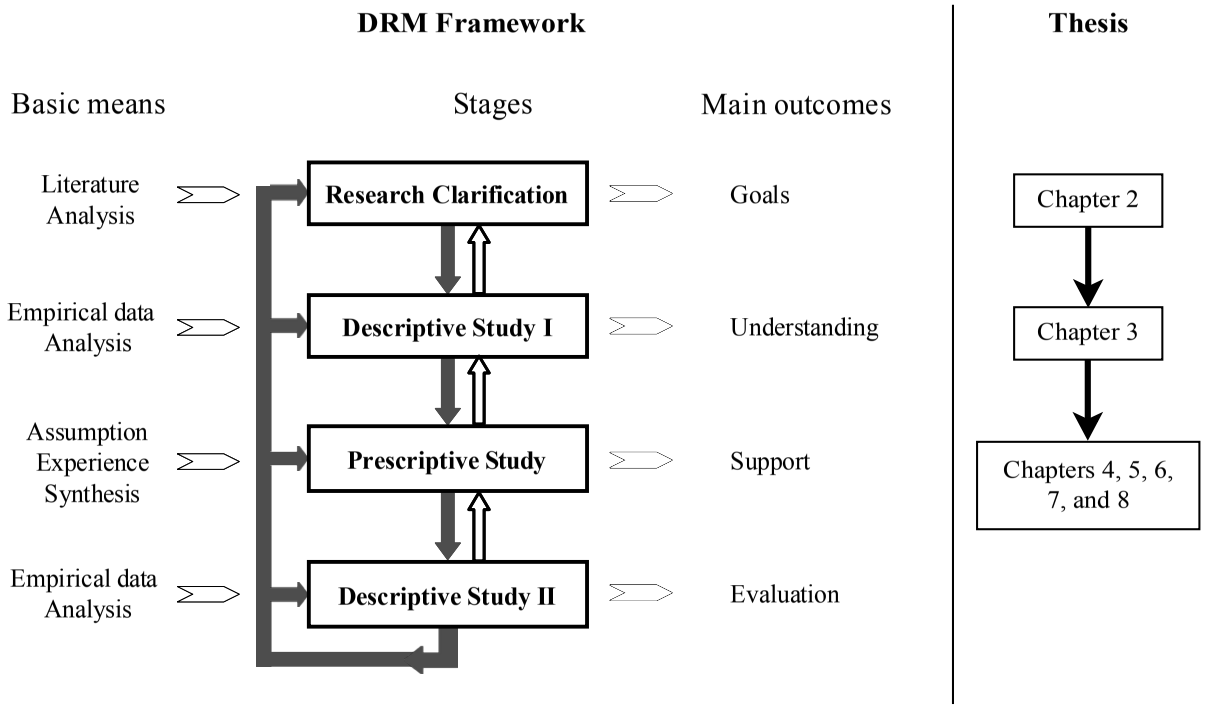

Figure 1.2: The design research methodology (DRM) framework [33] and thesis chapters.

1.5.2 Stage 2: Descriptive Study I

Following the previous stage, i.e., having a clear goal and focus, this stage is dedicated to reviewing the literature and observing the existing situation. During this stage, the researcher obtains a better understanding of the existing situation and provides the basis for the next stage of the DRM framework. Chapter 3 presents the Descriptive Study I stage in this thesis. Literature review and field study are used to obtain the goal of this stage.

- **Literature Review** A literature review is a comprehensive and critical analysis of published research literature, books, and other sources related to a particular topic or research question. It aims to provide an overview of existing knowledge and research on a specific subject, identify gaps in the literature, and highlight areas for further research [100].

  Reviewing the literature has become one of the foundations of evidence-based software engineering [67]. Systematic literature reviews, systematic mapping studies, and tertiary studies help present the current knowledge and findings of a specific topic. To investigate the terminology of software artefacts, we performed an analysis of secondary and tertiary research papers that classify technical debt types and provide definitions of the terms. We collected the initial set of papers using a search string and completed the set following the snowballing guidelines provided in [220]. There are alternative literature synthesis methods, such as *expert review* or *ad hoc literature selection* [100].

  We performed a literature review using snowballing (see Chapter 3 for more details) on the topic of technical debt as it is used to discuss the degradation of software artefacts. The technical debt (TD) metaphor deals with sub-optimal solutions that have a long-term impact on the system and its development. Investigating technical debt helps us reason about their degradation. Finally, we assume that if TD is important to study for a certain artefact, it might indicate that the artefact is important for the organisation the same way as assets are important for organisation.

  The analysis of secondary literature presented in this thesis belongs to stage two of the DRM model, i.e., Descriptive Study I. It is the follow up to the position paper presented in Chapter 2. We performed the analysis of secondary literature to find about the current state of research related to technical debt types as a proxy to assets.

- **Field Study**

  Field studies are the research that are conducted in a real-world, natural setting that study a specific phenomena. Field studies are appropriate for the cases where the researchers will not change any of the variables, i.e., there is no deliberate change or intervention from the researchers [196]. Researchers conduct field studies when they aim to develop a deep understanding of a phenomena in its setting, e.g., the organisation studied in this thesis.

  The objective of the field study was to identify the valuable artefacts, i.e., assets, for organisations from their perspectives. The data collected during the field study is used to perform a secondary analysis [179]. To identify the assets from the industrial perspective, we opted to perform a field study since it facilitates the study of the phenomena in its context and the researchers aimed to understand *what is going on* and *how things work*, i.e., exploration of the phenomena. We have followed the labelling guidelines provided by Saldaña [184] to label the collected data from the field study.

  We performed a field study (see Chapter 3 for more details) with five companies to identify and collect software assets from industry's perspective and gain an initial description of the current situation (Stage two of the DRM model). The field study is complementary to the systematic literature review. The collected data led to the creation of a taxonomy of software assets.

1.5.3 Stage 3: Prescriptive Study

This stage is dedicated to increase the understanding of the current investigated situation and elaborate on the desired situation. In this stage the researchers examine various different factors that impact the situation. During this stage the researchers examine different factors that impact the current situation. Chapters 4, 5, 6, 7, and 8 present case studies and a sample study used to obtain the goal of this stage.

- **Case Study**

  A case study is a research methodology that involves an in-depth investigation of a particular individual, group, organisation, event, or phenomenon. It is an empirical inquiry that is often used in social sciences, business, education, and other fields to examine complex real-life situations and to

gain a deep understanding of the underlying factors and dynamics at play. In a case study, the researcher collects and analyses a data sources such as interviews, observations, documents, and archival data to generate detailed and nuanced descriptions of the case being studied. The ultimate goal of a case study is to develop insights, theories, and recommendations that can inform future research, practice, and decision-making in the relevant field [180, 225].

Case studies are often observational, and the collected data is often investigated using qualitative analysis and quantitative analysis [180, 225]. A case study is "an empirical enquiry that investigates a contemporary phenomenon within its real-life context, especially when the boundaries between phenomenon and context are not clearly evident" [225, p. 13]. We chose to perform five industrial case studies to investigate architecture- and source-code-related assets. The objectives of the case studies were to explore and understand the factors that might impact quality degradation using technical debt as a proxy for code and architectural degradation in industrial context. The studies were conducted on a specific software components. We selected the cases by convenience, because of the availability of the cases and collaborating companies. We could retrieve certain data and information from these cases that is hard to retrieve from similar cases from the industry.

An alternative method for investigating the phenomenon is action research. In action research, the researcher engages with members of the organisation – in our case, industrial partners – to diagnose problems and implement interventions as potential solutions [84]. We opted to perform a case study since: (i) the nature the study was still exploratory; (ii) data is collected in a consistent manner; (iii) the research questions are answered by inferences from data; (iv) the data collected allows us to plan interventions, but they have been yet implemented in the studied organisations.

The results of the case studies help us understand the impact of quality degradation in different cases and components while considering different factors and each components' context. The results can be used to understand the degradation of source-code-related assets and can provide insights when managing assets and dealing with their degradation. We conducted the case studies (See Chapters 4, 6, 7, and 8) to understand impact of the degradation of certain assets and elaborate on the different factors that impact their degradation (stage three in the DRM framework, i.e., Prescriptive Study).

- **Sample Study**

  A sample study is a method that is performed over a certain population of actors, and it aims to achieve generalisability [196]. Sample study is one of the popular methods in software engineering research, and it is frequently used to study large sets of software development projects, in particular OSS projects [196]. We chose to perform a sample study to investigate the survivability of code technical debt items in an industrial system and 31 open-source systems from the Apache Foundation.

  Given the fact that sample studies might not provide depth of insight, alternative methods for investigating the phenomenon can be used. Such studies can be field studies or case studies that require more specific context [196]. However, field studies and case studies have their limitations. The limitations of the field studies are the following: It cannot provide statistical generalisability [196]. There is no control over events, and there is low precision on measurements [196].

  We performed a sample study to investigate the survivability of code technical debt items in industrial and open-source settings. The sample study presented Chapter 5 belongs to stage three of the DRM model, i.e., Prescriptive Study. And it is conducted to understand the impact of the degradation of certain assets.

1.5.4 Empirical Data Sources

The research done in this thesis is based on empirical evidence collected and synthesised in collaboration with five companies in the construction machinery, communication technology industry, banking and financial services and open-source systems. The research partner companies are Ericsson, Fortnox, Time People Group, Qtema, and Volvo CE. All the described companies work on software-intensive product development and are research partners in research projects (SERT[1] and SHADE). Table 1.2 summarises the empirical data source and data type used in this thesis and maps them to each chapter.

[1] See www.rethought.se

Table 1.2: Data sources and types used in this thesis.

Chapter	**Empirical Data Source [Data Type]**
Chapter 3	Academic Peer Reviewed Articles [Qualitative Data – Tertiary study of nine literature reviews]
	Five Companies (Ericsson, Fortnox, Time People Group, Qtema, and Volvo CE) [Qualitative Data - 413 Collected Notes and Statements]
Chapter 4	One Industrial Case - One system (2 MLOC) [Quantitative Data - Extracted from archival information of version control and static code analysis tool]
Chapter 5	Two Industrial Cases (2 MLOC) - 31 OSS Cases (3 MLOC) [Quantitative Data - Extracted from archival information of version control and static code analysis tool]
Chapter 6	One Industrial Case - 267 Components (15 MLOC) [Quantitative Data - Extracted from archival information of version control and static code analysis tool]
	One Company - Six Participants [Qualitative Data - Statements from Focus Group used for validation of results]
Chapter 7	One Industrial Case - Five Components (40.5 KLOC) [Quantitative Data - Extracted from archival information of version control and static code analysis tool]
	One Company - Six Participants [Qualitative Data - 30 Collected Notes and Statements and 10 Pages of Focus Group Transcriptions]
Chapter 8	One Industrial Case - 10 Components (132 KLOC) [Quantitative Data - Extracted from archival information of version control and static code analysis tool]
	One Company - Six Participants [Qualitative Data - Statements from Focus Group used for validation of results]

1.5.5 Validity

This section presents the validity of the results presented in this thesis. The studies conducted in this thesis have limitations which are discussed separately in each corresponding chapter. Here, we will focus on the *generalisability*, *replicability*, and *reliability* of the results of the thesis. The discussed validity is based on the criteria provided by Runeson et al. [181] and Yin [225].

- **Generalisability:** Generalisability is the extend to which the findings of the research can be generalised outside of the investigated cases and be applicable in other situations [181]. Ivarsson and Gorschek [93] present four aspects to evaluate research relevance, namely subjects, context, scale, and research methods. The methods used in this work investigate industrial cases in real-world context. The collected data is from the investigated systems and components. The subjects participating in this research are from different roles including: system architects, product owners, project managers, team leaders, enterprise architects, solution architects, and many more with the first-hand experience from the industry. Therefore, their insights are on par with the general population of industry professionals.

  The context and scale of our research are organisations that deal with software-intensive products and services. A representative sample would allow for the generalisation of the results. An understanding of the whole population is needed to precisely characterise the representativeness of the sample [163]. However, we cannot have a precise understanding of the population due to the sheer number of organisations dealing with software-intensive products and services. We tried to mitigate this thread by including different organisations and components in this research. Throughout this thesis, we use convenience sampling and cannot guarantee that all kinds of organisations are represented.

- **Replicability:** Replicability is the ability of obtaining the same results by other researchers using the data collected with different setups [38, 39]. The majority of the results presented in this thesis are context-specific and related to the companies investigated for each study. Therefore, a thread to validity of this work is the problem of replicating the same results in such context. In order to mitigate and limit this threat, the authors have provided meta-models, source-code for analysis and data collection, and detailed description of the collected data to provide sufficient information for replication of the results.

- **Reliability:** Reliability refers to the extent that same findings and conclusions can be obtained by different researchers if they follow the same procedure as the original study [225]. When conducting qualitative studies, the threat to validity is the replicability of the results and the process [124]. The work presented in this thesis are designed, conducted, and replicated by the authors. The results might be affected by their personal experience and biases. We tried to mitigate this threat by involving more than one author when collecting and analysing the data and drawing conclusions.

1.6 Research Contributions

This thesis contributes to the software engineering field by defining assets, the terminology in asset management, and types of asset degradation. A taxonomy of assets in software engineering is created and empirical evidence from industrial cases is presented. The thesis presents a model to evaluate degree of ownership and contribution alignment for a given component. Moreover, the thesis presents industrial case studies conducted to provide evidence on asset degradation by assessing factors that might impact its faster accumulation. Figure 1.1 summarises the contributions of the thesis. Each chapter of the thesis is mapped to a contribution, research gap, and a DRM stage. The details of the thesis contribution are presented below:

C1. Definition of assets in software engineering, the terminology in asset management, and types of asset degradation.

The first contribution of this thesis is providing the definition of assets in software engineering and its and related terminology such as asset degradation and its types. Moreover, a taxonomy of assets is created to identify an initial set of software assets. The thesis finding motivate the investigation on software assets and asset management in software engineering. This contribution is related to *G1.* and includes the papers in Chapters 2 and 3.

C2. Empirical evidence on asset degradation.

The second contribution of the thesis is providing empirical evidence from industrial cases illustrating asset degradation. This thesis provides four case studies and a sample study to investigate and provide empirical evidence on asset degradation. Despite many existing literature on quality

degradation on technical debt, evidence from industrial cases are scarce. The investigated cases in this thesis are from different settings and different companies. Each study considers different factors and illustrates how each factor might impact specific assets' degradation. This contribution is related to *G2.* and includes the papers in Chapters 4, 5, 6, 7 and 8.

C3. A model to evaluate the degree of ownership and contribution alignment.

One of the factors that impacts the degradation of assets is teams' structure and to what extend they own and contribute their code. In order to investigate whether the teams' ownership and contribution alignment impacts the degradation of code, a model was created. The existing metrics to evaluate developer contributions to code are one-dimensional, i.e., they only include one aspect. The presented model in Chapter 6 is created to calculate the degree of ownership and contribution alignment using multiple metrics. The model is designed to adopt based on the availability of the data, i.e., metrics can be added or removed from the model based on data availability. This contribution is related to *G2.* and includes the papers in Chapters 4, 5, 6, 7 and 8.

C4. Empirical evidence supporting and illustrating the impact of ownership and contribution alignment on asset degradation.

The thesis presents two case studies that have used the ownership and contribution alignment model (OCAM) to investigate and illustrate the impact of ownership and contribution alignment on asset degradation. This contribution is related to *G2.* and includes the papers in Chapters 7 and 8.

1.7 Discussion

Assets, other than those which have been extensively examined and studied, such as code, are an integral part of any software development organisations. Assets degrade over time. The continuous maintenance and evolution of software impacts the assets involved in the development process [114, 116].

The recurrent maintenance, and evolution causes assets to degrade. The degradation of assets needs to be managed since they have value for organisations (see Chapter 2).

Asset degradation is unavoidable due to the evolution of the software [114, 116], and all products' assets degrade [106]. Asset degradation comes in different forms: deliberate, unintentional, and entropy. Understanding asset degradation is crucial for managing assets' degradation and mitigating its impact. This is the first step to plan and execute maintenance activities to lessen the impact of asset degradation or possibly avoid the exponential growth of it (see Chapter 2)

One of the aspects that highlights the importance of understanding asset degradation is its eventual financial impact on the development of software-intensive products and services [14, 106]. The degradation of assets leads delays and difficulties in development activities as they increase the cost of change on assets [20]. The financial costs of degradation are a major concern and topic of research in the past years [9, 28, 136, 210]. Therefore, software-developing organisations take significant interest in understanding asset degradation and managing it.

Our research investigating software assets in industry led to the creation of a taxonomy of assets. The taxonomy provides an initial set of assets and asset types (see Chapter 3). The knowledge about degradation and the awareness of organisation's assets can support management activities [178].

Asset management activities increase the cost of software development. Software developing organisations can priorities management activities to target more severe degradation on any particular asset. To achieve this, assets need to be measured and monitored.

The state of assets' degradation can be assessed using different metrics. One of the metrics used to evaluate the degradation of assets is the technical debt (TD), e.g., code technical debt, as measured by industrial tools, to measure the degradation of code. TD has traditionally been used to measure the consequences of short-term decisions on long-term software development [14, 106]. The TD metaphor is widely spread, and has been studied by many researchers [178], and it has turned into a topic of interest for both academia and industry, and it has grown from a metaphor to practice [107]. Therefore, TD provides a good measure to assess the degradation of assets since it is extensively studied in the literature, it is a familiar concept to practitioners, and it has become a practice in industry [106, 178].

The utilisation of methods, metrics, and models to understand the factors impacting asset degradation is crucial. For example, technical debt is one of the common, well-studied, and in-practice metrics used by many companies to survey and investigate particular assets [30, 178]. Technical debt can be measured in different assets, for example, source code or tests. However, understanding the relevant contextual factors that might impact asset's degradation

will facilitate the interpretation of the causes of the degradation. For example, understanding the degree of contribution to source code can help interpret its degradation [210].

Throughout this thesis, we illustrate the impact of asset degradation on developing software-intensive products or services. The results suggest that the degradation can be the consequence of different factors, for example: i) how the accumulation of technical debt associated to different development activities (i.e., to new development, bug fixing, or refactoring) impact code degradation (see Chapter 4); ii) how degradation 'survives' (i.e., the survivability of code smells, bugs, and vulnerabilities as TD items); and how the misalignment between ownership and contribution impacts the faster accumulation of asset degradation (see Chapters 7 and 8). These studies illustrate the complexity of the asset degradation phenomena in software development and highlight the necessity to perform empirical studies to understand individual factors.

The overall takeaway is that understanding asset degradation is the necessary first step to plan sufficient asset management activities [15, 128]. These activities include proactive and reactive solutions to mitigate the impact of asset degradation. In order to achieve this, software developing organisations, together with the research community, need to identify software assets, raise the awareness about asset degradation, investigate the factors that impact the degradation of assets, and be able to manage the degradation.

1.8 Implications for Research and Practice

Here are the main implications for researchers and practitioners based on the findings of this thesis:

- In large organisations, the development of software-intensive products and services is closely intertwined with the social and organisational aspects of work. This is evidenced by the abundance of assets, such as *Business Models* and *Product Management Documentation*, that are linked to the social and organisational aspects of development. Therefore, there is a need to define and establish standards for these assets, including their perceptions and methods for measurement and monitoring.

- Organisation factors seem to have an impact on how assets degrade, therefore software development organisation have to consider these factors (e.g., who are responsible for a particular component) to optimise their asset

management processes and also to raise awareness about the importance of understanding ownership to avoid architectural knowledge vaporisation.

- We have studied a limited subset of assets, and a limited set of factors. Degradation might behave differently in other assets (e.g., in test code), and different roles (e.g., architects or testers) might behave differently when it comes to introduce or mitigate asset degradation, and this needs to be taken into account when planning asset management activities.

1.9 Conclusions and Future Work

This thesis presents the research conducted on software assets, and in particular, asset degradation. Our aspiration is to establish asset degradation in software engineering. We argue that understanding assets and asset degradation is vital to improving software development practices. To reach this goal, we have conducted multiple empirical studies. We included industrial data to strengthen the findings of the thesis. The research collaboration with five companies investigating more than 20 million lines of code has led to the results presented in this thesis.

The contributions of this thesis is two-fold: (i) defining and understanding assets and asset degradation; and (ii) providing evidence on the impact of different factors on the faster accumulation of asset degradation. We are aware that there are a plethora of assets and numerous factors impacting them. Given the wide scope of the topic and the challenges studying them in the limited time, we have focused on a particular, narrow scope, i.e., source code and architecture.

Studying and understanding asset degradation is challenging due to its large scope. There are many factors that impact the degradation of individual assets. And to further complicate the situation, is the impact of the degradation of assets on each other, i.e., the propagation of degradation. Finally, the overall impact of the degradation on reducing the pace of delivery of software, drives the research community and the industry to invest in management activities. The research presented in this thesis is the initial step to further investigation assets and asset degradation.

The long-term objective of that this thesis aspires to achieve is to promote asset management in software engineering and particularly in designing and developing software-intensive products and service. As the next steps, we plan the following:

- Our goal is to explore and identify assets' qualities are measured in industry, what tools they use to measure assets' characteristics and properties. We plan to perform studies to identify the metrics and tools used by the industry to support the taxonomy presented in Chapter 3.
- A natural progression of this work is to explore and investigate the propagation set degradation on dependent assets. As we have discussed in Chapter 2, asset degradation can propagate to other dependant, related assets. It is vital to understand the propagation of asset degradation in order to provide effective solutions and tools to mitigate its impact.
- The research presented in this thesis focuses on certain assets, i.e., source-code- and architecture-related assets. We plan to investigate assets that are less explored and studied such as, assets related to organisation.

Chapter 2

Assets in Software Engineering - What are they after all?

This chapter is based on the following paper:

> **Ehsan Zabardast**, Julian Frattini, Javier Gonzalez-Huerta, Daniel Mendez, Tony Gorschek, and Krzysztof Wnuk. "Assets in Software Engineering: What are they after all?" *In Journal of Systems and Software.* (pp. 111485) Elsevier (2022)[1]

[1] https://doi.org/10.1016/j.jss.2022.111485

During the development and maintenance of software-intensive products or services, we depend on various artefacts. Some of those artefacts, we deem central to the feasibility of a project and the product's final quality. Typically, these central artefacts are referred to as assets. However, despite their central role in the software development process, little thought is yet invested into what eventually characterises as an asset, often resulting in many terms and underlying concepts being mixed and used inconsistently. A precise terminology of assets and related concepts, such as asset degradation, are crucial for setting up a new generation of cost-effective software engineering practices.

In this position paper, we critically reflect upon the notion of *assets in software engineering*. As a starting point, we define the terminology and concepts of assets and extend the reasoning behind them. We explore assets' characteristics and discuss what *asset degradation* is as well as its various types and the implications that asset degradation might bring for the planning, realisation, and evolution of software-intensive products and services over time.

We aspire to contribute to a more standardised definition of *assets in software engineering* and foster research endeavours and their practical dissemination in a common, more unified direction.

Keywords: *Assets, Asset Management, Software Artefacts, Asset Degradation, Technical Debt*

2.1 Introduction

A fundamental challenge in producing software-intensive products and services is coping with the continuous changes that the business environments and the customers demand from the products [46]. These frequent changes have consequences on software artefacts, the main building blocks of the development process of software-intensive products and services [143].

A software artefact is defined as "a work product that is produced, modified, or used by a sequence of tasks that have value to a role [143]." This is a broad definition, and the number of artefacts produced or used along the development and maintenance activities of software-intensive products and services may be extensive. However, in our perception, software artefacts related to source code and architecture seem to be more prominent in literature and better understood than other artefacts like manuals or requirements, despite the latter having an at least comparable impact on the development life cycle.

Although all software artefacts are subject to quality degradation, it is neither practical nor efficient to exercise control of the quality for all of them. Quality control is strengthened by continuous use. In our perception, continuously controlling and ensuring quality is justified if an artefact is *intended to be used more than once* over time, i.e., if it is used several times during the development or maintenance activities. This, as we will argue, is one of the key characteristics of the related term *"asset"*. However, software engineering lacks a profound understanding of the term and its implications for software engineering practices.

Assets and asset management are popular terms used outside of software engineering. Reviewing the literature reveals that, despite the popularity of the term (see, e.g., [6, 186]), often related to tangible (physical) systems, it has not yet received much attention in software engineering. Nonetheless, we argue for the importance of asset management in software engineering as well as its evolution, and we postulate the importance of a well-defined vocabulary.

In this position paper, we critically discuss and characterise assets and extend the reasoning around related concepts such as asset degradation, i.e., the loss of value of an asset. We discuss various types of value degradation and the possible implications of these on the planning, realisation, and evolution of software-intensive products and services. Our contribution is rooted in our long-term academia-industry collaborations[2], and the empirical studies we conducted together with our industrial partners. The novelty of this work is three-fold:

- Characterising assets. Though there is a definition for assets (ISO 55000 [92]), there is no good way of distinguishing what an asset is and what it is not. Characterising assets can help identify them.

- Categorising the types of degradation. Discussing and defining different types of degradation can help practitioners and researchers better understand and investigate assets' degradation.

- Delineating from related concepts such as Technical Debt. As consequence of our continuous work and collaboration with the industry, we have observed that practitioners often have a different perspective on Technical Debt (TD) than academic authors. Other assets that go beyond code-related artefacts are of importance for practitioners and the industry is concerned with the consequences of their continuous use (which indicates that it might be worth keeping control over). Highlighting this industrial

[2]See, for instance, `www.rethought.se`.

perspective helps direct the research efforts to understand the needs of industry and validate our work.

Looking at related terms in software engineering literature, it seems that when referring to assets in software development, the term TD [14, 55] is becoming a *catch-for-all* that works for all and every negative consequence that may or may not happen to assets. TD can be a metric that can help us measure the degradation of assets. Yet, TD is often mistaken for actual degradation due to the lack of a concise definition of asset degradation. Moreover, TD has not yet considered the propagation of degradation [2], i.e. the degradation of certain assets stemming from the degradation of other, interrelated assets. Two examples of this propagation are i) the degradation of an architectural description element such as an activity diagram can lead to the degradation of source code, manifested as code-level TD; or ii) the propagation of code deprecation, which can propagate to the test-base, as in [198].

We conclude the paper with a discussion of future perspectives for research and practice to foster contemporary research endeavours in the community of software engineering researchers and practitioners in a common, more unified direction.

2.2 Assets in Software-Intensive Products and Services

In this section, we describe characteristics of assets, explore asset degradation and its propagation, and define asset management. We conclude this section by discussing our work in the context of software evolution.

2.2.1 Assets' Characteristics

An *asset* is a software artefact that is intended to be used more than once during the inception, development, delivery, or evolution of a software-intensive product or service. Assets have "potential or actual" value to the organisation [92]. Their value can be tangible or intangible, and financial or non-financial [92].

If an artefact is used once and discarded or disregarded afterwards, it does not qualify as an asset. In our understanding, an asset is, per definition, an artefact, but not necessarily the other way around (see Figure 2.1).

An artefact must be intended to be used more than once along the software development life cycle to be evolved and justify maintenance. However, this

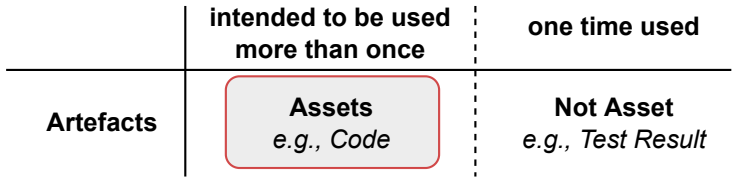

Figure 2.1: Difference between *assets* and *not assets*. Assets are artefacts that are intended to be used more than once along the software development life cycle.

intention might not be apparent to the involved stakeholders; hence even an artefact that was originally assumed to be used only once but later actually used more than once qualifies as an asset. If an artefact has value for the organisation, the implication is that there is a need for controlling the artefact's quality. One-time-used artefacts, as we argue, do not have any relevance over time, rendering maintenance and evolution inefficient. In the paragraphs below, we provide a few examples:

Artefacts that Qualify as Assets

- *Organisation Structure*: A document that summarises the roles involved in the development and their responsibilities in the organisation is an artefact that is intended to be used more than once because it is accessed whenever a specific competence needs to be located.

- *Activity Diagram*: A visualisation of the system's behaviour is an artefact that is intended to be used more than once if it is used or accessed during the design and the development phases.

- *Code*: Manually-written code is an artefact that is intended to be used more than once because it is used more than once. Each individual commit can be seen as a usage of the artefact.

Artefacts that Do Not Qualify as Assets

- *Generated View*: An automatically generated view (of parts) of the system that is only used once to capture the current state of the system is not an asset since it will be regenerated after the next system modification.

- *Meeting Notes*: Meeting notes are one-time-used artefacts when they are used to create properly formatted meeting minutes and afterwards discarded.
- *Test Result*: Test results can occur as one-time-used artefacts if they — when deemed important— are translated into a ticket or issue, while the test result artefact is discarded and not used again.

It is important to highlight that this qualification as an asset is context-dependent. What counts as an asset for an organisation depends on, to name a few aspects, what product or service they deliver, the organisation's ways of working, and which artefacts are reused and which are not. For example: in the context of a company where UML model views are automatically extracted from the code and never modified, the code qualifies as an asset, but the extracted views do *not* qualify as assets due to their singular use. On the contrary, consider a different company where the code is automatically generated from models through model transformations, and the code is never modified. In this second example, the code does *not* qualify as an asset since it is only used once, but the models and the model transformations responsible for code generation qualify as assets in that particular organisation.

2.2.2 Asset Degradation

Software engineering practices commonly imply making trade-offs in solutions of competing qualities. The option to go for the highest possible quality might lead to over-engineering the solution. Is it always worth deciding for the highest quality solution, or is a "good-enough" solution preferable (whatever this eventually is and however this might eventually be measured)?

When referring to software development and assets, the term "technical debt" comes to mind. However, TD research mainly focuses on code-related assets [178] and tends to depict the consequences of non-optimal design decisions [14]. Moreover, TD does not consider the propagation of "debt" from one artefact to other artefacts [2]. Rios et al. [178] discussed the three main reasons why TD is mainly studied on code-related assets:

i "The concept of TD was initially coined by Ward Cunningham with a focus on coding activities [..] and this, in some way, may have influenced the initial directions of research in the area. [178]"

ii "There is already a great amount of work that investigates the quality of software from indicators collected from the source code of the projects.

Tools already available for this purpose may have been used as the starting point of the research community to analyse how debt can affect software projects. [178]"

iii "The types of debt related to the code (architecture, design, code, defect, test) tend to cause effects that can be felt more quickly by the development team. [178]"

The degradation of certain assets, e.g., code, has been widely studied [178]. Such assets are more tangible and we can measure them to see how much they are deviating from the —often purely academic— gold standard.

We define degradation as the loss of value that an asset suffers due to intentional or unintentional decisions caused by technical or non-technical manipulation of the asset, or associated assets, during all stages of the product life-cycle. All assets can degrade, which will affect their *value* —for example, their usability [98]— in different ways.

We define three different types of degradation.[3] This classification was created as a result of discussions in a series of workshops with our industrial partner companies working on product development, including, for instance, Ericsson and Volvo CE, as part of a long-term academia-industry collaboration (`rethought.se`). We classify degradation as *Deliberate*, *Unintentional*, and *Entropy*:

- **Deliberate**: Taking a conscious, sub-optimal decision to accommodate short-term goals at the expense of long-term value. The asset is degraded as a result of a modification comprising a conscious non-optimal decision. We take a shortcut, knowing its consequences, and eventually pay its price.

- **Unintentional**: Sub-optimal decisions based on lack of diligence are unintentional forms of degradation. The asset is degraded as a result of a modification comprising an unconscious non-optimal decision. We are not aware of the other, better alternatives to our decision; therefore, we do not foresee the consequences of selecting an alternative. There is no awareness of the fact that the usability of the asset can be hindered.

- **Entropy**: According to Lehman, "as a software program is evolved its complexity increases unless work is done to maintain or reduce it" and "[it] will be perceived as of declining quality unless rigorously maintained

[3] Fowler presents a similar classification for the debt metaphor at `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`

and adapted to a changing operational environment" [114, 116]. This ever increasing complexity and perceived quality decline might encompass the degradation of the assets involved in the development of the software system. The degradation that is due to the continuous evolution of the software, and which is not coming directly from the manipulation of the asset by the developers is entropy. Entropy can introduce degradation in the form of natural decay due to technology and market changes on assets even when we have quality management mechanisms to avoid such degradation.

Here, we provide examples of asset degradation. The numbers in the list correspond to the numbers in Figure 2.2.

Deliberate Degradation:

1. **Deliberate Degradation of Organisation Structure**: If the organisation's structure (a team's constellation document) is over-simplified on purpose by associating only the main role to an individual, the asset is deliberately degraded. The saved effort comes at the expense of the structure not representing all access rights and responsibilities, hence inhibiting its use for use cases, where more than the main role of an individual needs to be retrieved.

2. **Deliberate Degradation of Activity Diagram**: If the activity diagram is drawn intentionally incomplete to save time (drawn quick-and-dirty), the asset is deliberately degraded.

3. **Deliberate Degradation of Code**: If shortcuts are taken to deliver the functionality by incurring TD on code, the asset is deliberately degraded.

Unintentional Degradation:

4. **Unintentional Degradation of Organisation Structure**: If the organisation structure, i.e., teams' constellation document, does not include unofficial roles of the employees, the asset is unintentionally degraded.

5. **Unintentional Degradation of Activity Diagram**: If outdated, archaic terminology is used to create an activity diagram (i.e., the current state-of-practice terminology is not used), the asset is unintentionally degraded.

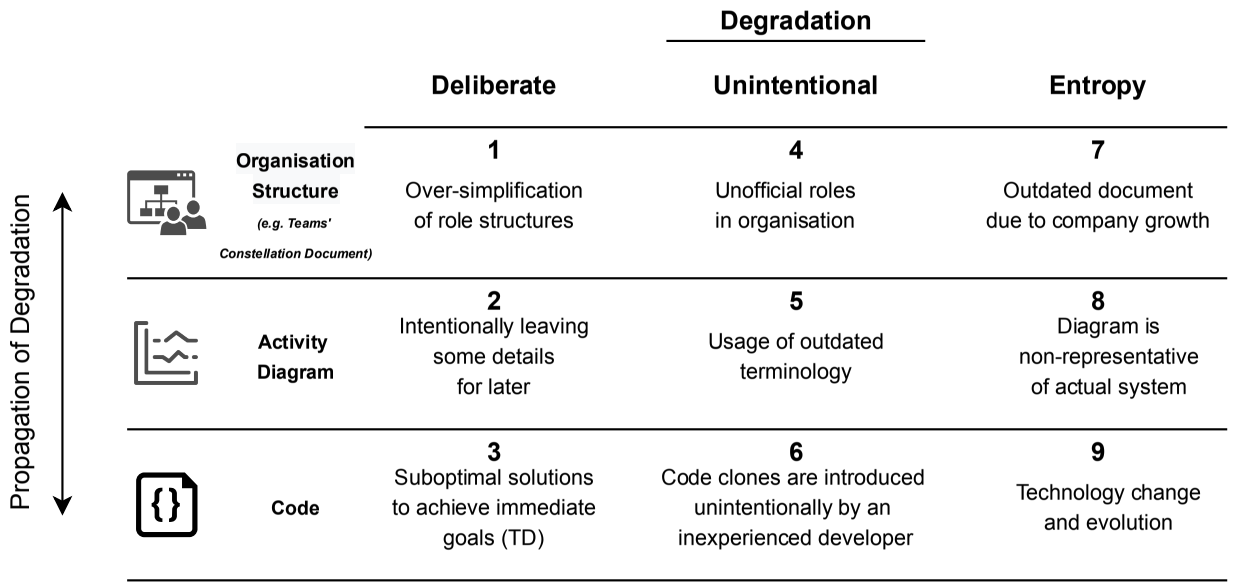

Figure 2.2: There are three sources for asset degradation: deliberate, unintentional, and entropy. The propagation of degradation causes other assets to degrade.

6. **Unintentional Degradation of Code**: If an inexperienced developer introduces code clones, the asset is unintentionally degraded.

Degradation Due To Entropy:

7. **Degradation of Organisation Structure Due To Entropy**: If the organisation structure, i.e., teams' constellation document, is not representative of the current structure of the organisation after the growth of the company, the asset is degraded due to entropy.

8. **Degradation of Activity Diagram Due To Entropy**: If the diagram is not representative of the current state of the system after changes have been made and implemented in the system, the asset is degraded due to entropy.

9. **Degradation of Code Due To Entropy**: If the quality of the code is impacted by releases of new versions of third-party libraries, the code can be outdated due to, for example, the usage of deprecated functions and the asset degraded due to entropy, not due to intentional or unintentional design decisions made by developers.

2.2.3 Assets' Degradation Can Propagate

When an asset is degraded, it is likely to influence the value of other dependent assets. We refer to this relation as "propagation" of asset degradation. It is important to note that degradation is different from interest. Interest is the term used in the TD metaphor defined as "the additional cost of the developing new software depending on not-quite-right code [14]." Interest is the inflation of the initial cost of repayment (principal) for one asset. At the same time, the propagation of degradation is the impact on other assets caused by a degraded asset.

An example can be the propagation of the degradation of the test base when we deprecate *dead* [4] versions of functions or services [198]. We can be tempted to deprecate these dead versions instead of removing them since they are never used. The rationale behind that decision is that deprecation will prevent developers from reusing them, provided that nowadays almost every development environment (IDE) will show deprecation information to developers. This might save us the cost of actually removing the code. However, these functions are

[4] We use here the concept of dead code to refer to versions of functions or services that are never used in the final product or by client applications.

still tested, and the test cases of the *dead*, deprecated functions or services can be reused as well in other tests, or even the deprecated functions or services can be called directly from integration or system tests to create the mock-up data or the required system state. In those cases, the deprecation information might not be visible for testers since sometimes the integration tests are using XML or other structured formats not supported by IDEs. Testers might continue reusing old, dead, deprecated versions of a function or service for some time, degrading the quality of the test base as a result of a coding design decision.

2.2.4 Asset Management

We define asset management as an umbrella term for the administration of assets and the activities that are related to creating and maintaining them as well as controlling their quality. Asset management considers the explicit control of assets throughout their life-cycle with a particular focus on the assets' degradation and "emendation", the conscious removal of degradation.

2.2.5 Assets and Software Evolution

Our work is reminiscent of the work of Lehman on software evolution where evolving (E-type) and not-evolving (S-type) systems are differentiated [114, 115, 116]. However, where previous work (e.g., [145, 177]) focuses on the external aspect of software evolution, and at a system level, our work focuses on the *internal* aspect (where assets are relevant). Figure 2.3 illustrates the differences of internal and external aspects in respect to the nature of software evolution, i.e., *quality* and *change* [177], and highlights how our work differs from already established literature:

- **Quality:** While the previous work on the topic mainly considers software evolution on the external quality of the software, i.e. toward the products' quality in use and customers; our work focuses on software evolution on the internal quality of software, toward the developers and the software development organisations.

- **Change:** The four *'Kinds of Software Change'* include *corrective, adaptive, perfective,* and *preventive* modifications [177]. Such modifications are discussed in the scope of software systems and products and in relation to the degradation of the external functionality. However, our work discusses the degradation of internal assets related to the development process.

To the best of our knowledge, despite the existing research on individual, case-based assets, there has not yet been given any proposal for a definition and characterisation of assets and asset degradation in software engineering.

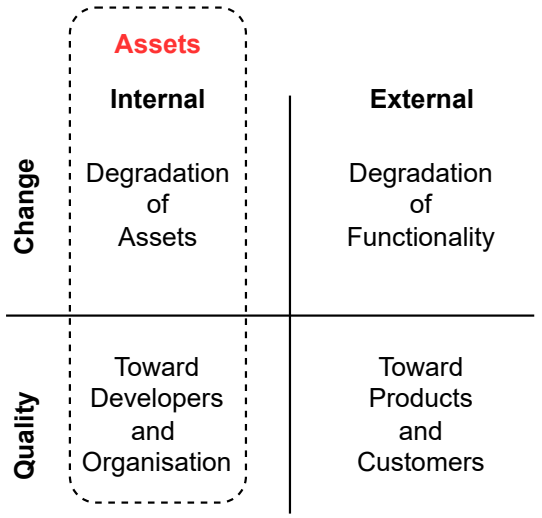

Figure 2.3: The nature of software evolution: Quality and Change [177]. Assets are within the internal aspect of software evolution.

2.3 Future Perspectives for Research and Practice

A shift in the view on assets, their value, and their management —starting with a precise and concise terminology— could enable a new way of considering how the metaphor of debt can be applied in software engineering and, related, building tangible asset taxonomies.

TD, as a metaphor, mapped from economic sciences —as it is predominantly used today— runs the risk of not properly reflecting the consequences of sub-optimal decisions. Therefore, it is limited in its application as a concept for capturing the effort necessary to remedy these consequences.

This is mainly due to the fact that the "interest" part of TD is often interpreted as covering all consequences of TD. At the same time, it reflects the

effort necessary to clear that particular TD in addition to the effort associated with its "principal". But since TD is confined to single assets, the complex relationship between various assets is omitted. As defined above, the notion of propagation is tailored to cover exactly this aspect and to better model the inter-dependencies of value-degradation between assets.

Research on traceability focuses on the relationships between certain assets, e.g., between requirements and code, or code and tests. Still, it struggles to provide a holistic view of assets and the relations among them. Rethinking the significance of assets and accordingly adjusting risk mitigation strategies is a first step towards rectifying this misalignment.

In particular, we would like to focus on the following research agenda and practical changes following this work:

- Effectively managing the quality of artefacts requires awareness and a profound understanding of which of those artefacts constitute (key) assets.

- As mentioned in Section 2.2.1, artefacts may qualify as assets without an apparent intention for reuse. This happens when the involved stakeholders fail to predict that an artefact will actually be used more than once. Failing to identify assets early on may cause additional cost, as – once the artefact has been correctly reassessed to be an asset – the quality of the asset has not been controlled and it degraded in the meantime. Predicting which eventually actually qualify as assets will be a major future research to reduce additional cost caused by lack of quality control.

- The quality of assets can degrade, thus, a depreciation of the value of assets takes place. While this notion of depreciation of software artefacts is not yet prevalent in Software Engineering (as it is in other disciplines, such as economics), it is nevertheless important to manage asset degradation. We hypothesise that there is a close relationship between the degradation of an asset and its depreciation. Asset degradation might increase maintenance costs, and hinder the ability to reuse these assets to create new functionality or services relevant for the users. We can intuitively link these two aspects (i.e., higher maintenance costs and lower reusability) to the depreciation of the degraded assets, however, this is again the subject of further research on the area.

- Research on degradation and propagation of degradation is a necessary foundation for controlling assets' quality.

- Once we understand the notion of asset degradation, we can effectively integrate practices for quality control (e.g., in continuous software engineering environments [101])

2.4 Conclusion

In this paper, we provide a definition for assets and asset management, introducing the concept of asset degradation. Based on our experience, after conducting workshops related to the concept of technical debt and asset degradation with several companies, we see that the term *technical debt* can be misleading. We may improve how we view and handle assets, particularly when discussing potential actions in practice, by introducing a coherent set of concepts and the corresponding, concise terminology, such as degradation.

We specifically aspire to construct a taxonomy for assets that provides the means for effective impact analysis to support evidence-driven risk management approaches properly. Those can then take: i) propagation of degradation from one asset to all inter-related assets, ii) measurement metrics for the degradation on the impacted assets, iii) evaluation of the severity of the propagated degradation to estimate the true consequences of degradation, iv) and strategies of emendation for managing the risk and clearing the degradation. Our vision is that this work provides a guideline for applying the concept of asset degradation in a practical context and better understanding the complex relationships between the value of assets in practice. At the time of writing this manuscript, we are working on creating a taxonomy of assets.

Chapter 3

A Taxonomy of Assets for the Development of Software-Intensive Products and Services

This chapter is based on the following paper:

> **Ehsan Zabardast**, Javier Gonzalez-Huerta, Tony Gorschek, Darja Šmite, Emil Alégroth, and Fabian Fagerholm. "A Taxonomy of Assets for the Development of Software-Intensive Products and Services" *In Journal of Systems and Software.* (pp. 111701) Elsevier (2023) [1]

[1] https://doi.org/10.1016/j.jss.2023.111701

Context: Developing software-intensive products or services usually involves a plethora of software artefacts. Assets are artefacts intended to be used more than once and have value for organisations; examples include test cases, code, requirements, and documentation. During the development process, assets might degrade, affecting the effectiveness and efficiency of the development process. Therefore, assets are an investment that requires continuous management.

Identifying assets is the first step for their effective management. However, there is a lack of awareness of what assets and types of assets are common in software-developing organisations. Most types of assets are understudied, and their state of quality and how they degrade over time have not been well-understood.

Method: We performed an analysis of secondary literature and a field study at five companies to investigate and identify assets to fill the gap in research. The results were analysed qualitatively and summarised in a taxonomy.

Results: We present the first comprehensive, structured, yet extendable taxonomy of assets, containing 57 types of assets.

Conclusions: The taxonomy serves as a foundation for identifying assets that are relevant for an organisation and enables the study of asset management and asset degradation concepts.

Keywords: *Assets in Software Engineering, Asset Management in Software Engineering, Assets for Software-Intensive Products or Services, Taxonomy*

3.1 Introduction

The fast pace of the development of software-intensive products or services impacts the decision-making process both for design and operational decisions. Such products and services are engineered by "applying well-understood practices in an organised way to evolve a product [or service] containing a nontrivial software component from inception to market, within cost, time, and other constraints. [102]"

Acting fast to cope with change can compromise the values of the delivered product, environment, development process, and the assets involved, such as source code, test cases, and documentation [45, 228].

Assets are software artefacts that are intended to be used more than once during the inception, development, delivery, or evolution of a software-intensive product or service [228]. Assets can lose their value and degrade "due to in-

tentional or unintentional decisions caused by technical or non-technical manipulation of an asset or its associated assets during all stages of the product life-cycle. [228]"

During the development of software-intensive products or services, decisions are usually made quickly to respond to change, focusing on fast delivery, leaving considerations on the assets involved as a secondary objective. Thus, these decisions often result in long-term negative consequences not only on the quality of the delivered product or service but also on the assets such as code, architecture, and documentation, to name a few.

Researchers and practitioners have traditionally used the technical debt (TD) metaphor [55] to refer to the impact of the intentional and unintentional sub-optimal decisions on assets as a consequence of meeting strict deadlines [106, 228]. In this work, we are investigating in assets' quality degradation; therefore, we explore the TD field as a representative form of quality degradation.

The motivation behind the work presented in this paper comes from industrial collaborations where the concept of TD did not fully cover their needs since code-based assets, although important, only represented a part of the challenge. From a practitioner's perspective, it is valuable to have the ability to identify what assets are available, which of them are important, and how and why they degrade. This is especially important as a product or service evolves over time, when assets are reused many times, and when they inevitably become subject to change and degradation. The degradation of assets is central to their maintenance and evolution [14].

In our previous work [228], we have defined *assets* and *asset degradation*. We have articulated the importance of identifying and studying them. From a research perspective, a potential benefit of introducing *assets* as a complementary concept to TD, with a broader definition, is that it takes all so-called "items of value" [141] into account and widens the view of what can hold value in the context of developing software-intensive products or services [228]. The widened definition also fosters an understanding of what assets can be negatively impacted by degradation and, thereby, the understanding that overlooking said assets can be detrimental to the development and evolution of such products and services. However, to gain such understanding, we must first understand what assets are subject to the concept, warranting the need for the taxonomy of assets presented in this work.

The paper is structured as follows: Section 3.2 provides the background and related work on the topic. Section 3.3 describes the research methodology, which separately covers the analysis of secondary studies and the field study. The re-

sults are presented in Section 3.4, together with the proposed asset management taxonomy. Section 3.5 discusses the principal findings and the implications of the results. The threats to validity are also discussed and addressed in Section 3.5. And finally, Section 3.6 presents the conclusion and the continuation of the work together with the future directions.

3.2 Background and Related Work

Assets related to software products or services have been studied previously. For instance, from a managerial perspective where the term asset is used to discuss how products in product lines are developed from a set of core assets with built-in variation mechanisms [159], or making emphasis only on business- or market-related assets, like the in the works by Ampatzoglou et al. [10], Cicchetti et al. [49], Wohlin et al. [221], and Wolfram et al. [222]. In contrast, in this work, our focus is on the inception, development, evolution, and maintenance of software-related assets.

3.2.1 Artefacts and Assets in Software Engineering

Describing how a software system is envisioned, built, and maintained is part of the Software Development Processes (SDP) [195]. The SDP prescribes the set of activities and roles to manipulate software artefacts, e.g., source code, documentation, reports, and others [45].

Artefacts in software engineering field have been traditionally defined as i) "documentation of the results of development steps" [45]; ii) "a work product that is produced, modified, or used by a sequence of tasks that have value to a role" [141]; and iii) "a self-contained work result, having a context-specific purpose and constitutes a physical representation, a syntactic structure and a semantic content of said purpose, forming three levels of perception" [141]. Software artefacts are, therefore, self-contained documentation and work products that are produced, modified or used by a sequence of tasks that have value to a role [141].

Understanding software artefacts, how they are structured, and how they relate to each other has a significant influence on how organisations develop software [45]. The documentation in large-scale systems can grow exponentially; therefore, there is a need for structurally organising software artefacts [45]. Artefacts defined by most of the SDPs are monolithic and unstructured [204]. The content of poorly structured artefacts is difficult to reuse, and the evolution

of such monolithic artefacts is cumbersome [190]. Therefore, different SDPs present various models for presenting software artefacts, e.g., the Rational Unified Process (RUP) [103, 104]. There are ways to classify and structure software artefacts based on well-known modelling concepts. Examples of such models are the work of Broy [45] and Silva et al. [190]. Moreover, there are ontologies and meta-models to classify artefacts in specific software development areas (e.g., Idowu et al. [89] Mendez et al. [142], Zhao et al. [232], and Constantopoulos and Doerr [52]).

However, the definitions of artefacts presented in the literature do not distinguish between:

i **Artefacts that have an inherent value for the development organisation** (i.e., an asset [228]) from the artefacts that do not have any value for the organisation[2] [228]. The value of each asset is a property that can characterise its degradation, i.e., if an asset degrades, although it continues being an asset, its value for the organisation has degraded.

ii **Artefacts that are intended to be used more than once** "An asset is a software artefact that is intended to be used more than once during the inception, development, delivery, or evolution of a software-intensive product or service... [228]" "If an artefact is used once and discarded or disregarded afterwards, it does not qualify as an asset [228]". We believe that the following example clarifies the distinction between *Temporary Artefacts* vs *Assets*: An API description used by developers as a reference has value in the development effort. If changes are made (new decisions, new ways to adhere to components, etc.), but the API description is not updated to reflect this, the utility (value) of the API description becomes lower (the asset degrades). On the other hand, an automatically generated test result is not seen as an asset, as it is transient or intermediate. It is generally created as a work product used once to be "transformed" into "change requests", "tickets" or other management artefacts. Once transformed into the new asset "change requests", it is discarded. New test results reports will be created (and discarded) on each execution of the tests. Therefore, artefacts that are intended to be used more than once and have value for the organisation (i.e., assets) need to be monitored by the organisation since their continuous maintenance and evolution renders the need for exercising quality control. [177, 228].

[2] Mendez et al. [141] define artefacts as having value for a role which is substantially different from having value for the organisation.

3.2.2 Asset Degradation and Technical Debt

In our previous work [228], we coined a concept *Asset Degradation* "as the loss of value that an asset suffers due to intentional or unintentional decisions caused by technical or non-technical manipulation of the asset, or associated assets, during all stages of the product life-cycle" [228]. All assets can degrade, which will affect their value for the organisation in different ways. Degradation can be deliberate, unintentional, and entropy [228]. *Deliberate degradation* is introduced by taking a conscious decision, understanding its long-term consequences and accommodating short-term needs. *Unintentional degradation* is introduced by taking a sub-optimal decision either because we are not aware of the other, better alternatives or because we cannot predict the consequences of selecting "our" alternative. Finally, *entropy* is introduced just by the ever-growing size and complexity that occurs when software systems are evolving [114, 116]. The degradation that is due to the continuous evolution of the software, and which is not coming directly from the manipulation of the asset by the developers, is entropy [228]. We argue that Technical Debt (TD), a metaphor introduced by Cunningham in 1992 [55] and which allows reasoning about the compromises resulting from sub-optimal decisions to achieve short-term benefits is one form of quality degradation. All assets (per definition artefacts, too) are subject to TD, i.e., while assets are created, changed, and updated, one might introduce TD on them [128]. We see the introduction of TD aligned with the three types of degradation mentioned above (i.e., deliberate, unintentional, and entropy) as we consider TD to be one form of asset quality degradation.

The TD metaphor has been extended and studied by many researchers [178]. It has been an interesting topic for both academia and industry, and it has grown from a metaphor to a practice [107]. TD is currently recognised as one of the critical issues in the software development industry [29]. TD is "pervasive", and it includes all aspects of software development, signifying its importance both in the industry and academia [106]. The activities that are performed to prevent, identify, monitor, measure, prioritise, and repay TD are called Technical Debt Management (TDM) [14, 85] and include such activities as, for example, identifying TD items in the code, visualising the evolution of TD, evaluating source code state, and calculating TD principal [178].

3.2.3 Taxonomies in Software Engineering

Scientists and researchers have long used taxonomies as a tool to communicate knowledge. Early examples are noted in the eighteen century, for instance,

the work of Carl von Linné [219]. Taxonomies are mainly created and used to communicate knowledge, provide a common vocabulary, and help structure and advance knowledge in a field [81, 111, 216]. Taxonomies can be developed in one of two approaches; top-down, also referred to as enumerative, and bottom-up, also referred to as analytico-synthetic [42]. The taxonomies that are created using the top-down method use the existing knowledge structures and categories with established definitions. In contrast, the taxonomies that use the bottom-up approach are created using the available data, such as experts' knowledge and literature, enabling them to enrich the existing taxonomies by adding new categories and classifications [215].

Software engineering (SE) is continually evolving and becoming one of the principal fields of study with many sub-areas. Therefore, the researchers of the field are required to create and update the taxonomies and ontologies to help mature, extend, and evolve SE knowledge [216]. The Guide to the Software Engineering Body of Knowledge (SWEBOK) can be considered as a taxonomy that classifies software engineering discipline and its body of knowledge in a structured way [37]. Software engineering knowledge areas are defined in SWEBOK, and they can be used as a structured way of communication in the discipline. Other examples of taxonomies in software engineering are the work of Glass et al. [82] and Blum [34]. Specialised taxonomies with narrower scopes are also popular in the field. These taxonomies are focused on specific sub-fields of software engineering such as Taxonomy of IoT Client Architecture [202], Taxonomy of Requirement Change [183], Taxonomy of Architecture Microservices [80], Taxonomy of Global Software Engineering [194], and Taxonomy of Variability Realisation Techniques [201] to name a few.

This paper presents a taxonomy of assets in the inception, planning, development, evolution, and maintenance of a software-intensive product or service. The taxonomy is built using a hybrid method, i.e., the combination of top-down and bottom-up. The details of the taxonomy creation are presented in Section 3.3.3.

3.2.4 Summary of the Gaps

In this paper, we identify and categorise assets in software development and software engineering. We use the concept of assets and their intentional and unintentional degradation. Moreover, we aim to address the following real-world problems:

- Bring awareness to practitioners and researchers. A precise and concise terminology of assets enables practitioners and researchers to consider new ways of dealing with asset degradation [228].
- The ripple effect that the degradation of an asset can impose on other assets is another aspect that necessitates the creation of taxonomy to understand relations between assets [106, 228].

In this paper, we identify the software artefacts that adhere to this definition of assets and are common in the industry. We aim to address the following gap: identifying and distinguishing assets by considering every aspect of software development. For example, assets related to*environment and infrastructure*, *development Process*, *ways of working*, and *organisational aspects* [14, 178] that have not received enough attention. We aspire to identify assets and provide a synthesis of existing knowledge in the area of asset management.

3.3 Research Overview

This section presents the research methodology of the paper. The process followed to build the taxonomy is divided into two main parts: a systematic analysis of secondary studies and a field study [196] of industrial cases using focus group interviews and field notes as data collection methods. Combining an SLR and a field study helps us look at the phenomenon from different perspectives and create a complete picture of the phenomena of assets. In addition, this multi-faceted view provides insights that can strengthen the validity of the results.

In this work, we are answering the following research question:

RQ : *What assets are managed by organisations during the inception, planning, development, evolution, and maintenance of software-intensive products or services?*

The rest of this section presents the analysis of secondary studies (See Section 3.3.1), the field study [196] (See Section 3.3.2), and taxonomy creation (See Section 3.3.3).

3.3.1 Analysis of Secondary Studies: Planning and Execution

This subsection describes the systematic analysis of secondary studies conducted in this work. All assets are subject to degradation. Asset degradation can be

classified into three categories deliberate, unintentional, and entropy [228]. Asset degradation can also be measured and monitored using different metrics, e.g., the amount of TD. TD is a familiar concept for practitioners and has received a growing interest in academia and the industry [178]. TD has become a broad research area that has focused on different assets. The fact that TD research studies a particular asset might imply that that asset might be of value for software development organisations, or at least that might be perceived as such by TD researchers. Therefore, we do believe we can use the TD literature as a proxy for identifying and categorising different types of assets. Since we are interested in assets' quality degradation, we explore the TD field as a representative form of quality degradation.

In order to study the state of art, we performed an systematic analysis of secondary studies to capture the classifications and definitions of the various assets addressed by these previous research works (top-down method) [42]. In our literature review, the goal was to identify systematic literature reviews, systematic mapping studies, and tertiary studies. We reviewed secondary and tertiary research papers by performing an SLR using snowballing, following the guidelines by Wohlin [220]. We selected snowballing as a search strategy as it allowed us to explore the area as well as its reported efficiency [16].

The starting set of papers for snowballing was collected through a database search in Google Scholar in October 2021 using the following search string in: *"technical debt" AND ("systematic literature review" OR "systematic mapping study" OR "tertiary study").* We chose to use the search string only in Google scholar since it is not restricted to specific publishers [16], and it can help avoid publisher bias [220]. We selected, with the inclusion and exclusion described below, the articles that presented a classification for TD, i.e., articles that present different types of TD.

The execution procedure to identify assets included the following steps as illustrated in Figure 3.1. The results of this process are presented in Section 3.4.1.

- **Step 1:** Collection of the start set of relevant articles (seed papers), including SLRs, SMSs, and tertiary studies on TD, by using a search string.
- **Step 2:** Evaluate the papers - start set in the first iteration - for inclusion/exclusion based on the criteria.
  - **Inclusion Criteria:**
    - The selected papers should report secondary or tertiary studies;

Figure 3.1: The execution process of the analysis of the secondary studies.

    - the papers should present any classification of TD and assets affected by TD;
    - the papers should be written in English; the papers' main aim is the literature review.
  - **Exclusion Criteria:**
    - The papers that present informal literature reviews and are duplication of previous studies are not included.

- **Step 3:** The snowballing procedure for identifying additional secondary and tertiary studies on TD that satisfy our inclusion/exclusion criteria:
  - **Step 3.1:** Backward snowballing by looking at the references of the selected papers. The backward snowballing was finished in one round.
  - **Step 3.2:** Forward snowballing by looking at the papers that cite the selected papers. The forward snowballing was finished in one round.
- **Step 4:** Extracting different *types of assets* and *assets* together with their respective definitions from the selected articles.
- **Step 5:** Synthesising the definitions of the *types of assets* and *assets* provided by the selected articles.
- **Step 6:** Creating the matrix of *types of assets* and *assets* based on TD classifications defined by the selected articles.

3.3.2 Field Study (Focus Group Interviews): Planning and Execution

To study the state of practice, we performed field studies, using focus groups and field notes as data collection in five companies to find evidence on how assets are defined and used. The five companies were selected using convenience sampling, as the companies are involved in an ongoing research project that focuses, among other topics, on addressing asset degradation challenges.

The focus groups' process is presented in Figure 3.2. The reports from the focus groups were coded and used for the construction of the taxonomy. We used the bottom-up method [42] for updating the existing structure that we obtained from the literature review. The focus groups were planned as a half-working day (four hours).

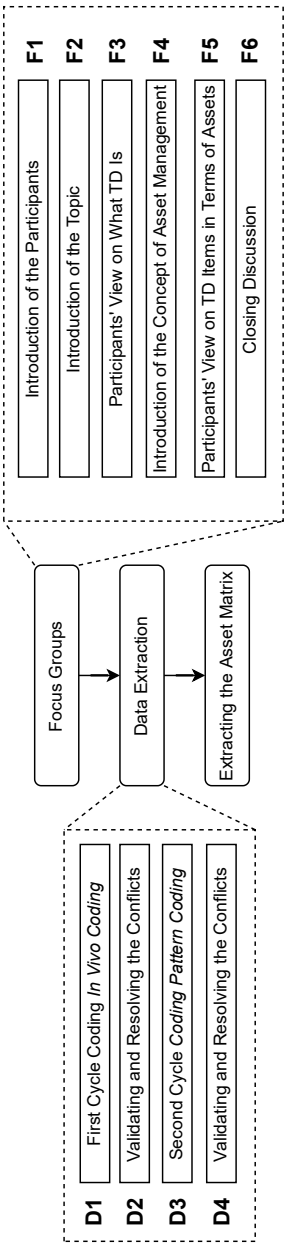

Figure 3.2: The field study execution process.

Case Company Characterisation

We have collected the data for this study by collaborating with five companies that work in the area of construction machinery, communication & ICT, consultancy services, and financial services. The research partner companies are Ericsson (telecommunication & ICT), Fortnox (Financial Services), Qtema (Consultancy Services), Time People Group (Consultancy Services), and Volvo CE (Construction Machinery).

The partner companies are mature in their development practices and have well-established, successful products. They are interested in continuously improving their products and development processes, which turns into their willingness to participate in studies like this. All the collaborating companies work on developing software-intensive products or services and are involved in a large ongoing research partnership[3]. The details of the case companies are presented in Table 3.1. Note that the order of the companies in Table 3.1 does not correspond with the order of focus groups (focus group IDs) in Table 3.4, which has been shuffled to preserve confidentiality.

Focus Groups' Procedure

The steps taken during the industrial focus groups are presented in this section. The focus groups include six steps (see Figure 3.2):

- ***F1.*** Focus group participants introducing themselves.
- ***F2.*** One of the moderating researchers presenting the topic.
- ***F3.*** Focus group participants discussing the topic, providing insight into their views/experiences with TD and document them in notes.
- ***F4.*** Focus group participants discussing assets and asset management in detail after a second presentation of the concept.
- ***F5.*** Focus group participants discussing what they wrote before as TD examples and rethinking them in terms of assets, asset degradation, and asset management.
- ***F6.*** A closing discussion and focus groups.

[3]See `www.rethought.se`.

Table 3.1: Case company details. The table is ordered alphabetically based on the name of the companies, and does not correspond to the order in Table 3.4.

Company	Domain	Investigated Site	Enterprise Size	Participants' Roles	Number of Participants
Ericsson	Telecommunication & ICT	Karlskrona, Sweden	Large	Senior System Architect Corporate Senior Strategic Expert Operations & Testing	3
Fortnox	Finance	Växjö, Sweden	Large	Head of Development Product Owner Development Manager System Architect Testing	14
Qtema†	Consultancy	Stockholm, Sweden	SME	Chairman of the Board Requirements Analyst Sales Manager Project Manager IT Administration Manager	6
Time People Group†	Consultancy	Stockholm, Sweden	SME	Data Consultant Project Manager Consultant Senior Agile Coach IT Project Manager Team Leader Chief Executive Officer (CEO) Consultant Test Leader	7
Volvo CE	Construction Machinery	Gothenburg, Sweden	Large	Enterprise Architect Solution Architect Business Information Architect	5

† Time People Group and Qtema participated in the same workshop.

Each Focus group starts with participants introducing themselves with background information about their work, including their current role in the organisation (step F1). One of the moderating researchers then presents the Focus group's agenda and covers the importance of the topic and the growing interest in value creation and waste reduction both in academia and in the industry (step F2).

After the initial introduction of the topic by the moderating researchers, the participants are divided into groups. They are asked to list and discuss the challenges with their ways of working (while considering varying aspects of TD), i.e., the problems they know or have encountered or experienced (step F3). After the time is up, the notes are read, discussed, and abstracted to a more general description and later put on a whiteboard. The connections between the items on the board are identified and marked down with a marker.

After a second presentation, i.e., introducing the participants to the concepts related to asset management and asset degradation (step F4), the participants add new items to the previous notes on the board. Participants then refine the items from the board for the rest of the Focus group (step F5). The Focus group ends with a closing discussion on the topic and the items (step F6).

In the context of this research, we have moved the focus from the traditional TD metaphor to asset degradation. In this framework, we talk about asset degradation as the deviation of an asset from its representation. That way, we can focus, potentially, on any type of asset and its representation. This framework provides us with a broader, holistic view that allows us to study how an asset's degradation (e.g., requirements) might introduce degradation in other assets (e.g., code or test cases).

The researchers' minutes that were written during each session were then aggregated and summarised in a report sent back to the participants, that were asked to provide us with feedback. The written notes from the participants and the final reports were used as raw data for creating the taxonomy using the data extraction method described in Section 3.3.2. The raw data (i.e., participants' notes and the reports) were used for coding and later to extract *types of assets* and explicit *assets*. The details of the data extraction and taxonomy creation are described in Section 3.3.3.

Unfortunately, since this research is under non-disclosure agreements (NDA) with participating companies, we cannot disclose any further information about the companies, the participants, or the collected materials.

Data Extraction

To create the matrix of assets from industrial insights, we use the hybrid method of coding, as suggested by Saldaña [184]. The coding is divided into two main cycles: First Cycle Coding and Second Cycle Coding.

First Cycle Coding of the raw data happens in the initial stage of coding. The raw data, which can be a clause, a sentence, a compound sentence, or a paragraph, is labelled based on the semantic content and the context in which it was discussed during the Focus group. We have used the *in vivo coding* method [184] to label the raw data in the first cycle. In vivo coding prioritises the participants' opinions [184]; therefore, it is suitable for labelling raw data in the first cycle coding in our study, where participants' opinions are used as input. It adheres to the "verbatim principle, using terms and concepts drawn from the words of the participants themselves. By doing so, [the researchers] are more likely to capture the meanings inherent to people's experiences." [197, p. 140] It is commonly used in empirical and practitioner research. [51, 78, 197]

The coding was done by two researchers independently (step D1, see Figure 3.2). The labels were then compared to validate the labels and to identify conflicting cases. A third researcher helped to resolve the conflicts by discussing the labels with the two initial researchers (step D2, see Figure 3.2).

Second Cycle Coding is done primarily to categorise, theorise, conceptualise, or reorganise the coded data from the first cycle coding. We have used Pattern Coding [184] as the second cycle coding method. Pattern codes are explanatory or inferential codes that identify an emergent theme, configuration or explanation [184, p. 237]. According to Miles et al. [148, p. 86], pattern coding is used in cases where: (i) the researchers aim to turn larger amounts of data into smaller analytical units. (ii) the researchers aim to identify themes from the data. (iii) the researchers aim to perform cross-case analysis on common themes from the data gathered by studying multiple cases.

Similar to the first cycle coding process, pattern coding was done by two researchers independently (step D3, see Figure 3.2). The results were compared to validate the classifications and to identify conflicting cases. A third researcher resolved the conflicting cases in a discussion session with the two researchers (step D4, see Figure 3.2). The results of the insights gathered from industrial focus groups are presented in Section 3.4.2.

3.3.3 Taxonomy Creation

To describe a precise syntax and the semantics of the different concepts used for the taxonomy creation, we created a metamodel (presented in Figure 3.3). The metamodel illustrates the structural relationships between the metaclasses, i.e., concepts, in the taxonomy. The metaclasses presented in the metamodel are:

- The *"AssetsTaxonomy"* metaclass is the container metaclass for the items in the model.
- The "TypeOfAsset" metaclass represents the hierarchical classification of assets. The items belonging to this metaclass can be further broken down into subclassifications representing various groups of assets. The types of assets are containers for the assets. Types of assets are identified from the state-of-the-art (i.e., existing academic literature), state-of-practice (i.e., the industrial insights gathered through the industrial focus groups), or the identified by researchers.
- The *"Asset"* metaclass represents assets. Each asset belongs to one and only one type of asset, assuring orthogonality by design. Assets are identified from the state-of-the-art (i.e., existing academic literature), state-of-practice (i.e., the industrial insights gathered through the industrial focus groups), or identified by researchers.
- The *"Reference"* metaclass represents the references from which each asset or type of asset has been identified. Reference can originate from academic literature (the literature review) or industrial insights (gathered from industrial focus groups). References can be mapped to individual *assets/type of assets* or multiple *assets/type of assets*.

The creation of the taxonomy included three steps. First, we summarised the relevant topics on TD types (top-down approach) with items that we extracted from the literature review. We created the matrix based on the literature's definitions, i.e., by synthesising the definitions to identify similarities, differences, and hierarchies of identified items. We have grouped the definitions provided by the literature based on their semantic meaning.

In the second step, we utilised the extracted assets from industrial focus groups (Section 3.3.2) to create a second asset matrix (bottom-up approach). Like the previous step, we used the definitions of the assets and their types and the participants' statements from the focus groups.

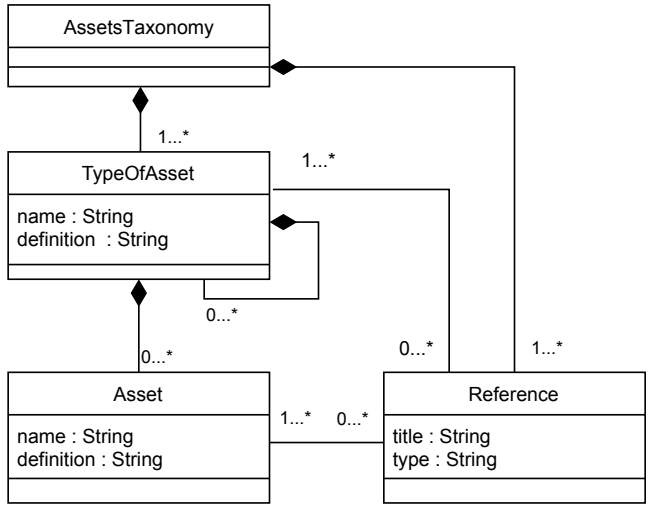

Figure 3.3: Asset Management Metamodel.

After the creation of the matrices, all the candidate assets in each matrix were listed and presented to all the researchers that were part of the industrial workshops. The researchers discussed whether each candidate asset and asset type should be included in the taxonomy. The discussion included ensuring each candidate asset adhered to two criteria: 1) the candidate asset should adhere to the definition of an asset, and 2) how a candidate asset would degrade. We decided to include a candidate after all the researchers agreed on it adhering to the criterion. The rest of the candidates were discarded.

By combining the two matrices in the last step, we created the asset management tree. To complete the tree, we added some nodes based on the researchers' expertise and extensive industry knowledge from decades of industrial research that we perceived were missing nodes and leaves. We mention such cases as *Author Defined Assets* (ADA) when presenting the results.

The process of adding ADA started with researchers suggesting assets that should be included in the taxonomy. These suggested assets were brought up in internal workshops[4] where all the researchers discussed, reflected, improved, and added or removed the suggested assets. *User Stories*, as an example, were suggested by one of the researchers to be considered as assets during one of

[4]Internal workshops refer to the workshops where the researchers discussed the taxonomy.

the internal workshops. The discussion was regarding (i) whether or not *User Stories* are assets, (ii) if they fit in the taxonomy according to the definition, (iii) where they belong in the tree, (iv) what they represent, (v) and how they degrade. After the discussions, the researchers decided that *User Stories* belong to *[AM1] - [AM1.1] Functional-Requirements-Related Assets* in the taxonomy tree. The assets included in the taxonomy should adhere to the definition of an asset [228].

3.3.4 Taxonomy Validation

This section describes the taxonomy validation procedure. According to Usman et al. [216], the validation process includes orthogonality demonstration, benchmarking, and utility demonstration. The taxonomy was created to be orthogonal by design (i.e., each element can only be a member of one group), as described in Section 3.3.3. Therefore, in this validation we focus on benchmarking and utility demonstration.

We conducted three separate workshops (one for each company) to validate the taxonomy and its structure with the six participants (including Product Owner, Developer, Software Architect, Scrum Master, Test Quality Assurance, and Research Engineer) from three companies namely, Ericsson (2 participants), Fortnox (3 participants), and Volvo CE (1 participant) who were involved during the industrial workshops for the data collection. The procedure for the validation workshops is presented below.

1. The taxonomy was sent to the participants to study before the validation workshops.
2. During the validation workshops, the taxonomy was presented to the participants.
3. After the presentation, the participants filled in a questionnaire that included four questions:
   (a) "*Select the assets from the taxonomy that you were not aware of.*"
   (b) "*Select the assets that you think should not be in the taxonomy.*"
   (c) "*Write down the assets that are missing from the taxonomy.*"
   (d) "*Prioritise the top 5 assets based on your experience. (index starting from 1 as the most important asset). You can add missing assets from the previous question as well.*"

4. At the end of each workshop, a discussion session was held where participants asked questions regarding the taxonomy and its content. The discussions were recorded.

We decided to apply changes to the taxonomy after the validation workshops were concluded, in cases when the majority of the participants suggested a given change in the taxonomy. The results of the validation workshops together with the discussion on the results are presented in Section 3.4.4.

3.4 Results

This section presents the results of the systematic literature review and then the results from the field study. We will present the asset management taxonomy built by aggregating the results from both. Finally, we will present the results of the validation workshops.

3.4.1 Results from the Analysis of Secondary Studies

The final list of selected papers included nine articles, presented in Table 3.2. To create the final assets' matrix (presented in Section 3.4.3), we extracted the data from each article. For example, in paper *P4* [128], we refer to Figure 8 on page ten of the article, where the authors summarise the "TD classification tree" and their respective definitions. The authors define Requirements TD as "the distance between the optimal requirements specifications and the actual system implementation, under domain assumptions and constraints" [128, p. 9]. Requirements is also mentioned as a TD item in *P3*, *P5*, *P6*, and *P8*. Table 3.3 presents the summary of our findings based on the types of TD. It is important to mention that the columns P1 to P9 in Table 3.3 retain the exact extracted words from the data. The "Emerging Category(ies)" and "Phase, during which the artefact is produced" columns are the synthesised information for each row. The codes of the papers are used as a reference throughout this paper. It is important to mention that Table 3.3 does not represent an overview of the final assets but only the ones extracted from the SLR. However, this matrix helps us to categorise assets and types of assets according to existing classifications.

Looking at Table 3.3 we can see that there are fewer categories in the earlier studies (*P1* and *P2*) compared to later studies (*P3* to *P9*). The more recent papers that follow these studies break down the bigger categories into more specific categories. For example, *Architecture and Design* are put into one category

Table 3.2: The articles gathered for the literature review during the snowballing process.

Code	Title	Seed Paper	Backward Snowballing	Forward Snowballing
P1	A Consolidated Understanding of Technical Debt [206]		•	
P2	An Exploration of Technical Debt [207]		•	
P3	Towards an Ontology of Terms on Technical Debt [5]	•		
P4	A Systematic Mapping Study on Technical Debt and Its Management [128]	•		
P5	Identification and Management of Technical Debt: A Systematic Mapping Study [4]	•		
P6	Managing Architectural Technical Debt: A Unified Model and Systematic Literature Review [30]	•		
P7	A tertiary study on technical debt: Types, management strategies, research trends, and base information for practitioners [178]	•		
P8	A systematic literature review on Technical Debt prioritization: Strategies, processes, factors, and tools [118]			•
P9	Investigate, identify and estimate the technical debt: a systematic mapping study [25]			•

in *P1* and *P2* and later, they are broken down into their own categories. It is important to mention that *P7* has fewer categories since the study is on the specific topic of *Architectural Technical Debt*.

3.4.2 Results from the Field Study

There were a total of four focus groups, each held with the participation of employees from different companies. The focus groups' procedure stayed the same, while the closing discussion of each focus group was on the topic of interest for that focus group's participants (the stakeholders). The topics included, but were not limited to, *Lack of Knowledge/ Competence*, *Architecture Lifecycle*, *Business Models for Products*, and *Backlog Update Issues/Backlog Size*.

Two researchers used the in vivo coding method to label 386 statements during the first cycle coding. After matching and validating the labels, 14 cases of conflicting labels were identified. The conflicting labels were resolved during a discussion session with a third researcher. The researchers agreed on the new labels for the conflicting cases during that discussion.

The focus groups are presented in chronological order in Table 3.4, i.e., *WS1* was the first focus group. It is important to mention that the columns WS1 to WS4 in Table 3.4 retain the exact extracted words from the data. The "Emerging Category(ies)" and "Phase, during which the artefact is produced" columns are the synthesised information for each row. Examining Table 3.4, we observe that assets are mentioned more often than types of assets in the industrial focus groups, whereas types of assets are more frequent in the literature review (see Table 3.3). Finally, assets that are related to *Operations*, *Management*, and *Organisational Management* were highlighted more in the industrial focus groups than the literature review.

3.4.3 The Asset Management Taxonomy

Using the key concepts extracted from the labelled data (presented in Table 3.3 and Table 3.4), we build the taxonomy of assets. The taxonomy contains the assets identified both through the literature review and through the industrial focus groups. The assets included in the taxonomy are presented in a tree (graph). The nodes represent the assets (the leaf nodes) and the types of assets (non-leaf nodes).

The tree presented in Figure 3.4 contains only the types of assets (The full tree is presented in Appendix 3.6). Note that the nodes in the tree in Figure 3.4 are mapped to represent their source. For example, a node can be assigned with

Table 3.3: Asset matrix from technical debt literature.

P1 2012	P2 2013	P3 2014	P4 2015	P5 2016	P6 2018	P7 2018	P8 2019	P9 2020	Emerging Category(ies)	Phase, during which the artefact is produced
● Features		● Requirements	● Requirements	● Requirements ● Usability	● Requirements ● Usability		● Requirements	● Requirements ● Usability	● *Product Requirements* ● *Quality Requirements*	Requirements
● Design\ Architecture ● Documentation	● Design and Architecture ● Documentation	● Architecture Design ● Documentation ● Design Specifications	● Architecture Design ● Documentation	● Architecture Design ● Documentation	● Architecture Design ● Documentation	● Architecture	● Architecture Design ● Documentation ● Architectural Documentation	● Architecture Design ● Documentation	● *Architecture Design Decisions Documentation* ● *Product Documentation* ● *Architectural Documentation*	Design
● Code	● Code	● Code ● Build ● Service	● Code ● Build ● Versioning	● Code ● Build ● Service ● Versioning	● Code ● Build ● Service ● Versioning	● Code	● Code ● Build ● Versioning	● Code ● Build ● Service ● Versioning	● *Source Code* ● *Build Documentation* ● *Web Services* ● *Versioning*	Development
● Testing ○ Defects	● Testing	● Test ○ Defects ● Test Automation	● Test ○ Defects	● Test ○ Defects ● Test Automation ● Test Case Documentation	● Test ○ Defects ● Test Automation		● Test ○ Defects	● Test ○ Defects ● Test Automation	● *Functional Test* ● *Test Automation* ● *Test Documentation*	Verification and Validation
● Infrastructure	● Environment ● Hardware Infrastructure ● Operational Processes	● Infrastructure	● Infrastructure	● Infrastructure	● Infrastructure		● Infrastructure	● Infrastructure	● *Environment & Infrastructure* ● *Environment & Infrastructure* ● *Operations*	Operations
		● Process		● Process ● Documentation Specifications	● Process			● Process	● *Process Managmenet* ● *Documentation Internal Rules & Specifications*	Management

Color Guide: ● Assets ● Types of Assets ○ Temporary Artefacts

Table 3.4: Asset matrix from industrial input.

WS1	WS2	WS3	WS4	Emerging Category(ies)	Phase, during which the artefact is produced
● Contradictory Requirements*	● Requirements*	● Requirements*	● Requirements *	● *Product Requirements*	Requirements
	● Documentation ● Architectural Documents ● Architecture	● Architectural Models ● Software Structure	● Documentation ● Architectural Documentation ● Architecture	● Documentation ● Product Documentation ● *Architectural Documentation* ● Architectural (Source Code)	Design
● Dangerous Code	● Code ● APIs ● Libraries	● Code ● API Versions ● Third Party Products	● Code	● Source Code ● *APIs* ● *Libraries\ External Libraries*	Development
	● Test Cases ● Automated Tests ○ Bug Reports	● Tests	● Test Cases	● *Test Cases* ● *Test Automation Scripts*	Verification and Validation
	● Kubernets	● Containers \Kubernets	● Application Data	● *Application Data* ● *Tools*	Operations
	● Ways of Working ● Coding Standards ● Product Roadmap ● Backlog	● Ways of Working ● Coding Standards ● Architectural Rules ● Product Management	● Documentation about Ways of Working ● Documentation Standards	● *Documentation about Ways of Working* ● *Coding Standards* ● *Architectural Internal Standards* ● *Documentation Internal Rules \Standards* ● *Product Management* ● *Product Backlog*	Management
	● Organisation's Roadmap	● Business Models	● Holistic Strategy ● Organisation's Structure	● *Organisation's Strategy* ● *Organisation's Structure* ● *Business Models*	Organisational Management

Color Guide: ● Assets ● Types of Assets ○ Temporary Artefacts

* The term *requirements* refers to software requirement area. We use requirement artefact when referring to documents such as SRS.

[P1] as a reference for an article in the literature review. Similarly, *[WS1]* as a reference for an asset coming from industrial focus groups[5]. And finally, Author Defined Assets (*[ADA]*) are assets included in the taxonomy by the researchers.

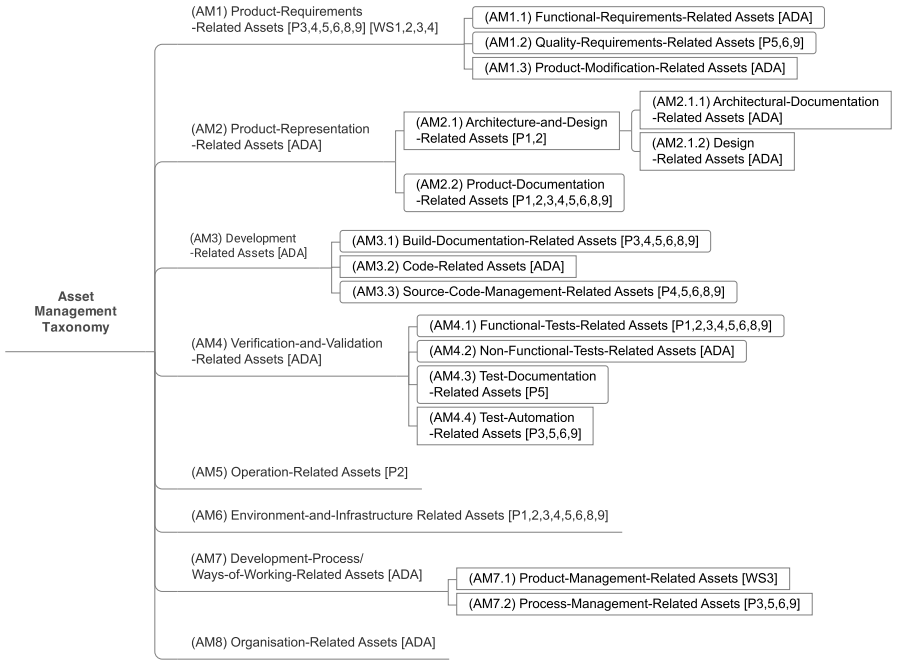

Figure 3.4: The Asset Management Taxonomy. The tree contains only the types of assets. The full tree is presented in Appendix 3.6.

The process described in Section 3.3.3, combining the asset matrices from the literature review, the input from the industrial focus groups, and completing the tree with *Author Defined Assets* (ADA) resulted in the taxonomy containing 24 types of assets and a total of 57 assets.

The eight main types of assets included in the taxonomy are:

- **Product-Requirements-Related Assets (AM1)** refer to assets (and types of assets) concerned with software requirements, including the elic-

[5]The IDs on the focus groups on Table 3.4 have been obfuscated to preserve anonymity and have no relationship with the order of companies shown in Table 3.1.

itation, analysis, specification, validation, and management of requirements during the life cycle of the software product.

- **Product-Representation-Related Assets (AM2)** refer to the assets (and types of assets) concerned with system and architectural design and any documentation related to these assets.
- **Development-Related Assets (AM3)** refer to the assets (and types of assets) concerned with the development of the software product, including the code, build, and versioning.
- **Verification-and-Validation-Related Assets (AM4)** refer to assets (and types of assets) concerned with software testing and quality assurance and the output provided by such assets that help the stakeholders investigate the quality of the software product.
- **Operations-Related Assets (AM5)** refer to assets (and types of assets) concerned with the data produced or collected from operational activities, e.g., any data collected during the use of the product or service.
- **Environment/Infrastructure-Related Assets (AM6)** refers to assets (and types of assets) concerned with the development environment, the infrastructure, and the tools (including support applications) that facilitate the development or deployment process.
- **Development-Process/Ways-of-Working-Related Assets (AM7)** refer to assets (and types of assets) concerned with product and process management and all the interrelated processes and procedures during the development process.
- **Organisation-Related Assets (AM8)** refer to assets (and types of assets) concerned with organisations, such as team constellation, team collaborations, and organisational governance.

In the remainder of the section, we present the eight major types of assets labelled *AM1-AM8*. We include the definitions of each type of asset together with their corresponding assets. Assets' definitions are presented in Appendix 3.6.

Product-Requirements-Related Assets (AM1)

Product-Requirements-Related Assets include the following three types of assets (see Figure 3.5):

Figure 3.5: Product-Requirements-Related Assets Sub-tree.

- *Functional-Requirement-Related Assets (AM1.1)* refer to the assets related to the functions that the software shall provide and that can be tested [37]. We have identified the following assets belonging to this type: *Feature-Related Backlog Items*, *User Stories*, and *Use Cases*.

- *Quality-Requirement-Related Assets (AM1.2)* refer to the assets related to non-functional requirements that act to constrain the solution [37]. We have identified the following assets belonging to this type: *System Requirements*, *User Interface Designs*, *Quality Scenarios* (i.e., the -ilities), and *User Experience Requirements*.

- *Product-Modification-Related Assets (AM1.3)* refer to assets that mandate a change of the system and, but not necessarily, the requirements. *Change Requests* is an asset we identified belonging to this type.

Product-Representation-Related Assets (AM2)

Product-Representation-Related Assets include the following two types of assets (see Figure 3.6):

- *Architecture-and-Design-Related Assets (AM2.1)* refer to the assets that are used to design, communicate, represent, maintain, and evolve the software product, which is divided into:

  - *Architectural-Documentation-Related Assets (AM2.1.1)* refer to the assets used to design, communicate, represent, maintain, and evolve the architectural representation of a software product. We have identified the following assets belonging to this type: *Architectural Models* and *Architectural Documentation*.

  - *Design-Related Assets (AM2.1.2)* refer to the assets that belong to the design that occurs during the development process. We have identified the following assets belonging to this type: *Design Decisions Documentation* and *System Designs*.

- *Product-Documentation-Related Assets (AM2.2)* refer to the assets that belong to the product documentation and the process of creating such documentation. We have identified the following assets belonging to this type: *Documentation Automation Scripts* and *Product Documentation*.

Figure 3.6: Product-Representation-Related Assets Sub-tree.

Development-Related Assets (AM3)

Development-Related Assets include the following three types of assets (see Figure 3.7):

- *Build-Documentation-Related Assets (AM3.1)* refer to the assets related to the build system itself, the build environment, and the build process. We have identified the following assets belonging to this type: *Build Plans*, *Build Results*, and *Build Scripts*.

- *Code-Related Assets (AM3.2)* refer to the assets that are related to the source code. We have identified the following assets belonging to this type: *Source Code*, *Code Comments*, *APIs*, *Architecture (Code Structure)* —i.e., a set of structures that can be used to reason about the system including the elements, relations among them, and their properties [19]—, and *Libraries/External Libraries*.

- *Source-Code-Management-Related Assets (AM3.3)* refer to the assets related to managing the source code, such as versioning and problems in code versioning and burndown charts. *Versioning Comments* is an asset we identified belonging to this type.

Verification-and-Validation-Related Assets (AM4)

Verification-and-Validation-Related Assets include the following four types of assets (see Figure 3.8):

- *Functional-Tests-Related Assets (AM4.1)* refer to the assets related to testing the functionality of the system, its related features, and how they work together. We have identified the following assets belonging to this type: *Unit Tests*, *Integration Tests*, *System Tests*, and *Acceptance Tests*.

- *Non-Functional-Test-Related Assets (AM4.2)* refer to the assets related to testing the quality attributes of the system and whether they satisfy the business goals and requirements. We have identified *Non-Functional Test Cases* as an asset which belongs to this type.

- *Test-Documentation-Related Assets (AM4.3)* refer to the assets related to documenting the testing process. *Test Plans* is an asset we identified belonging to this type.

Figure 3.7: Development-Related Assets Sub-tree.

Figure 3.8: Verification-and-Validation-Related Assets Sub-tree.

- *Test-Automation-Related Assets (AM4.4)* refer to the assets that are utilised for automated testing of the system. We have identified the following assets belonging to this type: *Test Automation Scripts* and *Test Automation (Real/Synthetic) Data*.

Operations-Related Assets (AM5)

Operation-Related Assets are all the assets created as the result of operational activities, extracted during the operational activities, or used during the operational activities, e.g., any data collected during the use of the product or service (see Figure 3.9). The operations-related assets include *Customer Data*, *Application Data*, and *Usage Data*.

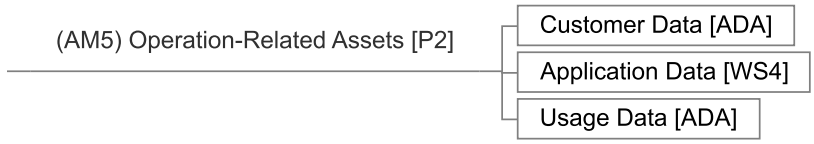

Figure 3.9: Operation-Related Assets Sub-tree.

Environment-and-Infrastructure-Related Assets (AM6)

Environment-and-Infrastructure-Related Assets are all the assets used in the development environment or as an infrastructure for development during software development (see Figure 3.10). The environment-and-infrastructure-related assets include *Deployment Infrastructure*, *Tools*, and *Tools Pipelines*.

Figure 3.10: Environment-and-Infrastructure-Related Assets Sub-tree.

Development-Process/Ways-of-Working-Related Assets (AM7)

Development-Process / Ways-of-Working-Related Assets include the following three types of assets (see Figure 3.11):

- *Product-Management-Related Assets (AM7.1)* refer to the assets related to the management of the product or service. These assets come from different stages, such as business justification, planning, development, verification, pricing, and product launching. We have identified the following assets belonging to this type: *Product Management Documentation*, *Documentation About Release Procedure*, *Product Business Models*, *Product Roadmap*, *Product Scope*, and *Product Backlog*.
- *Process-Management-Related Assets (AM7.2)* refer to the assets related to managing the development process, including internal rules, plans, descriptions, specifications, strategies, and standards. We have identified the following assets belonging to this type: *Requirements Internal Standards*, *Architectural Internal Standards*, *Documentation Internal Rules / Specifications*, *Build Internal Standards*, *Coding Internal Standards / Specifications*, *Versioning Internal Rules / Specifications*, *Testing Internal Rules / Specifications / Plans / Strategies*, *Process Internal Descriptions*, *Process Data*, and *Documentation About Ways of Working*.

Organisation-Related Assets (AM8)

Organisation-Related Assets are all the assets that represent organisations' properties. The identified organisation-related assets include Organisational Structure, Organisational Strategy, and Business Models (see Figure 3.12):

3.4.4 Summary of Validation Workshops

In this section, we present the summary of the taxonomy validation workshop carried out with industrial practitioners, as described in Section 3.3.4.

The industrial participants were asked to select the assets from the taxonomy that they were not aware of or assets that were new to them. The participants collectively selected 33 assets. These assets and the number of times they were chosen are presented in Table 3.5. The participants were asked to select the assets that, based on their experience, should not be in the taxonomy. Overall, the participants selected 13 assets. These assets and the number of times there were chosen are presented in Table 3.6.

Figure 3.11: Development-Process/Ways-of-Working-Related Assets Sub-tree.

Table 3.5: This table summarises the assets that the validation workshop participants were not aware of. The number represented the number of times the asset was selected by the participants.

	Asset	Selected by
1	**AM2.2** - Documentation Automation Scripts	4
2	**AM7.1** - Product Management Documentation	3
3	**AM1.2** - Quality Scenarios	2
4	**AM3.1** - Build Plans	2
5	**AM7.1** - Product Business Models	2
6	**AM7.2** - Requirements Internal Standards	2
7	**AM7.2** - Architectural Internal Standards	2
8	**AM7.2** - Documentation Internal Rules / Specifications	2
9	**AM7.2** - Build Internal Standards	2
10	**AM7.2** - Versioning Internal Rules / Specifications	2
11	**AM7.2** - Testing Internal Rules / Specifications / Plans / Strategies	2
12	**AM7.2** - Process Internal Descriptions	2
13	**AM7.2** - Process Data	2
14	**AM1.**1 Feature-Related Backlog Items	1
15	**AM1.2** - User Experience Requirements	1
16	**AM2.1.1** - Architectural Models	1
17	**AM3.1** - Build Results	1
18	**AM3.2** - Architecture (Code Structure)	1
19	**AM3.2** - Libraries / External Libraries	1
20	**AM3.2** - Web Services	1
21	**AM3.3** - Versioning Comments	1
22	**AM4.1** - Acceptance Tests	1
23	**AM4.4** - Test Automation (Real / Synthetic) Data	1
24	**AM5** - Usage Data	1
25	**AM6** - Deployment Infrastructure	1
26	**AM6** - Tools	1
27	**AM7.1** - Documentation About Release Procedure	1
28	**AM7.1** - Product Scope	1
29	**AM7.1** - Product Backlog	1
30	**AM7.2** - Coding Internal Standards / Specifications	1
31	**AM7.2** - Documentation About Ways of Working	1
32	**AM8** - Organisation's Structure	1
33	**AM8** - Organisation's Strategy	1

Figure 3.12: Organisation-Related Assets Sub-tree.

Table 3.6: This table summarises the assets that the validation workshop participants, based on their experience, deemed not needed to be included in the taxonomy. The number represented the number of times the asset was selected by the participants.

	Asset	Selected by
1	**AM1.2** - Quality Scenarios	1
2	**AM3.1** - Build Plans	1
3	**AM3.2** - Libraries / External Libraries	1
4	**AM4.1** - Unit Tests	1
5	**AM5** - Usage Data	1
6	**AM7.1** - Product Management Documentation	1
7	**AM7.2** - Requirements Internal Standards	1
8	**AM7.2** - Architectural Internal Standards	1
9	**AM7.2** - Documentation Internal Rules / Specifications	1
10	**AM7.2** - Build Internal Standards	1
11	**AM7.2** - Versioning Internal Rules / Specifications	1
12	**AM7.2** - Testing Internal Rules / Specifications / Plans / Strategies	1
13	**AM8** - Business Models	1

In an open question, the participants were asked to write down the assets that were missing from the taxonomy. *Git Comments*, *Code Review Data*, *Contract Documentation*, and *Communication Channels* were mentioned in the questionnaire. Finally, we asked the participants to prioritise the top five assets based on their experience. The answers to the last question are summarised in Table 3.7.

The utility of the taxonomy is demonstrated through the use of taxonomy by practitioners and classification of existing knowledge [40]. The participants in the validation workshops were able to use the taxonomy to identify and classify assets.

There are no existing taxonomies to compare the classification schemes; therefore, the taxonomy benchmarking was done with experts' knowledge. The validation workshop participants did not make any suggestions regarding the structure of the taxonomy. All participants agreed that the schema and categories (i.e., types of assets) were adequate.

The participants in the validation workshops selected the important assets based on their experience. The six most selected assets are *[AM1] - Product Requirements Related Assets*, *[AM3] - Development Related Assets*, *[AM6] - Environment and Infrastructure Related Assets*, *[AM7] - Development Process / Ways of Working Related Assets*, *[AM2.1.2] - Design Related Assets*, and *[AM4] - Verification and Validation Related Assets* (see Table 3.7). The selected important assets come from different asset types illustrating the importance of all assets. This emphasises the need to identify and maintain all assets in the organisation.

Table 3.7: This table summarises the assets that the validation workshop participants, based on their experience, selected as the top five assets. The number represented the number of times the asset was selected by the participants.

Asset	Selected by
AM1 - Product Requirements Related Assets	4
AM1.1 - Functional Requirements Related Assets	1
AM1.3 - Product Modification Related Assets	1
Use Cases	1
System Requirements	1
AM2 - Product Representation Related Assets	1
AM2.1.2 - Design Related Assets	2
Product Documentation	1
AM3 - Development Related Assets	3
AM3.2 - Code Related Assets	1
Source Code	1
Architecture	1
AM4 - Verification and Validation Related Assets	2
AM4.1 - Functional Tests Related Assets	1
AM5 - Operation Related Assets	1
AM6 - Environment and Infrastructure Related Assets	3
AM7 - Development Process / Ways of Working Related Assets	3
AM7.1 - Product Management Related Assets	1
AM8 - Organisation Related Assets	1

3.5 Discussion

This section discusses our findings in light of the research question, followed by the general lessons learned and implications.

3.5.1 Principal Findings

RQ : What assets are managed by organisations during the inception, planning, development, evolution, and maintenance of software-intensive products or services?

In Section 3.4.3 we have presented a taxonomy of assets, which includes eight major types of assets *AM1* to *AM8*. Although the taxonomy is orthogonal by design (i.e., an asset or a type of asset can only be classified as a member of one type of assets), assets and types of assets are not isolated, i.e., some assets and types of assets are interrelated. For example, *architectural documentation* is directly related to *architecture* since architectural documentation is a representation of the architecture of the system.

During internal workshops and the taxonomy creation procedure, presented in Section 3.3.3, we identified several meta-characteristics of assets. These meta-characteristics are:

- **Easier to contextualise**: It is easier for the stakeholders to identify such assets in the software product context. For example, the data that the company acquires from the operation of the product, i.e., *Application Data*, can be used as input to improve the product.
- **More tangible**: Some assets are more prominent in industry and have been studied and discussed more deeply before, and therefore the asset is not alien anymore. For example, every software company, one way or another, has *Code*, in one form or another, or a *Product Backlog* with specific characteristics which is familiar to all people involved in the development of the software-intensive products or services.
- **Easier to measure**: There are already existing metrics used to measure the state of such assets. For example, there are many metrics available to measure *Source Code*, such as LOC and Cyclomatic Complexity.
- **Used universally**: The assets that are defined in the same way across different organisations and academia, meaning that they are not organisation-specific. For example, the *software's architecture (Code Structure)* is a universal and inherent aspect of any software-intensive product or service.

Out of the eight major types of assets, two types have been studied more extensively, namely *Development-Related* Assets and *Product-Representation-Related* Assets. These results are not new and have been highlighted in previous studies such as [14], i.e., prior studies on TD focus on source-code-related assets. These types of assets are easier to study due to the abundance of metrics and evaluation methods and, therefore, have been studied in many research articles. The reason behind this might be that:

- The TD metaphor was initially introduced in the context of prevalent assets [14]. Therefore the researchers have spent more time investigating and exploring this specific phenomenon. For example, many papers investigate a software product's architecture, exploring different ways of evaluating architecture using different tools and measurements.
- These types of assets are easier to contextualise in the TD metaphor, i.e., identifying such assets and how they can be subject to incur debt. For example, the concept of code smells is easier to grasp since it is a more tangible artefact. It is simple to define how the software product can incur debt if the code does not align with certain "gold standards"; i.e., it is smelly.

The rest of the types of assets have not received extensive time to be explored. The reason behind this might be that:

- These types of assets were added later as "types of technical debt," such as Requirements Debt [65, 119] and Process Debt [138]. The TD metaphor was not initially used to deal with these types of assets [14]. These types of TD were introduced in an effort to extend the metaphor and, therefore, have not been investigated thoroughly.
- Unlike the other types (i.e., *Development-Related* and *Product-Representation-Related* Assets), it is harder to identify and/or define how and to what extent one can incur debt in software products. For example, incurring Documentation Debt might differ in different companies and development teams.

We have seen that the existing literature on TD classifies various TD types and presents ontologies on the topic. These classifications have evolved since the introduction of the extended TD metaphor. We observe that the relevant asset categories we have extracted from industrial insights can be mapped to the classifications provided in TD literature. We observe that:

- Some existing TD types and categories, such as code that are well-defined and well-recognised, fit into similar categories as in the presented taxonomy.
- Some types of assets that are relevant to the industry have been understudied or not even studied at all. There is room for extending the research in such areas [178], e.g., *Operations-Related Assets (AM5)* and *Environment-and-Infrastructure-Related Assets (AM6)* (see Section 3.4.1).
- By creating the taxonomy, we highlight both the areas of interest and the gaps in research. Therefore, identifying the areas in the software engineering body of knowledge that need to be investigated and the areas that need to evolve according to the current interest.

Finally, our taxonomy of assets has an innate relationship with the TD research and the taxonomies, ontologies, secondary, and tertiary studies in the TD topic since as mentioned in Section 3.3.1, TD is one form of asset quality degradation. The taxonomy of assets is created using empirical evidence from peer-reviewed articles and co-production with industrial practitioners. The taxonomy of assets provides actual tangible assets, whereas other studies mainly present areas where assets belong to. Asset degradation goes beyond TD and considers different types of degradation, including the ripple effect and chain reactions of degradation that stems from the relations between assets. TD is a part of a bigger problem, i.e., managing asset degradation when companies deal with software-intensive products or services [228]. And there is a need to consider a more holistic view to be able to address the asset degradation problem.

The synthesis of the results of taxonomy validation shows that the variety of processes followed by the participants' organisations is what makes some asset types well-known for some participants while seeming alien for others. The taxonomy can help widen practitioners' understanding regarding assets, by making some practitioners aware of assets that they are not used to. Moreover, we observe that the assets in AM7 were selected more often than the other asset types (see Table 3.5). The results of the validation step solidify the need for the creation of the taxonomy.

3.5.2 Lessons Learned

This section presents the lessons learned from running the industrial focus groups, synthesising the findings, and creating the taxonomy. The importance

of source-code-related assets is undeniable (i.e., assets in *AM1*, *AM2*, *AM3*, and *AM4*). However, we observe that the social and organisation aspect of the development is very important to the industry, through these aspects have not received as much attention in the TD area [14, 178], although the TD community has already identified them as a research area that deserves more attention [138]. Taking a look at some statements from participants in the industrial focus groups highlights this fact. Examples of such statements are:

- *"There are many people who work in the same area in the same code base. Creates conflicts and slow releases."*
- *"The problem is the delta operation, and the plan is at such a high level that it is impossible to understand. Too abstract."*
- *"... training the teams in what is considered best practices improves team cohesion and eases collaboration."*
- *"[There is] no holistic platform strategy (Conway's law)."*

The large-scale software projects developed in large organisations are highly coupled with the social and organisation aspect of work. The prevalence of assets related to the social and organisation aspect of development, e.g., *Business Models* and *Product Management Documentation*, indicates the necessity to characterise and standardise such assets, how they are perceived, and how they are measured and monitored.

While creating the taxonomy, we observed that assets do not exist in isolation, i.e., they are entities with characteristics and properties that exist in a software development environment. In the following, we will discuss the assets with similar characteristics and properties and assets that have implicit relations with each other.

Assets that have similar characteristics and properties. For example, *Unit Tests* have similar characteristics and properties as *Source Code*, i.e., unit tests are code, and therefore, their value degradation can have analogous connotations. This means that there are possibilities to evaluate such assets' degradation with similar characteristics and properties using similar metrics. Still, the degradation of one asset (e.g., Source Code) might impact or even imply the degradation of the other asset (e.g., Unit Tests) [2]. Therefore, such coupling and relations of the assets should be considered when analysing and managing such assets.

Assets that have implicit relations between each other. Implicit relations between assets can arise from their inherent coupling properties. Different assets related to certain aspects of the product will have implicit relations that are not visible in the taxonomy as presented now. For example, *Architectural Models* and *Architecture (Code Structure)* have an inherent relationship. Architectural Models should be the representation of the architecture of the system, i.e., the code structure. Therefore, similar to the previous point, the value degradation of the assets with such implicit relations can have analogous connotations. Their degradation might impact the degradation of the other related assets. For example, the degradation of any of the functional-requirements-related assets will eventually be reflected in the degradation of functional-test-related assets.

3.5.3 Contributions

The contribution of this work is the following:

- Providing common terminology and taxonomy for assets that are utilised during software development and;
- Providing a mapping over the assets and the input used to create the taxonomy, i.e., input from the literature and input from the industry

One contribution of the taxonomy is that it is a guideline for future research by providing a map of different types of assets. The map illustrates the different areas defined and studied and those that lack a shared understanding or are under-explored. Therefore, the taxonomy provides a summary of the body of knowledge by linking empirical studies with industrial insights gathered through the industrial focus groups. Providing a common taxonomy and vocabulary:

- Makes it easier for the different communities to communicate the knowledge.
- Creates the opportunity to find and build upon previous work.
- Helps to identify the gaps by linking the empirical studies to the taxonomy.
- Highlights potential areas of interest.
- Makes it possible to build and add to the taxonomy (new assets, details) as knowledge is changed over time by researchers in the field.

- For practitioners, the evolving taxonomy can be used as a map to identify assets and the degradation that can impact other assets (ripple and chain effects that can spread the degradation to other assets like degradation of code causing degradation on test-code or vice-versa [2]), where such implicit degradation are not immediately detectable.

Finally, it might help large organisations to deal with developing software-intensive products or services that rely on external resources to help them achieve the business goals of their products. A major external contributor to new knowledge that can help practitioners in the industry is research findings. Therefore, understanding and applying the research findings is crucial for them. Having a taxonomy of assets summarising the state-of-the-art and state-of-practice body of knowledge for the assets utilised for developing software-intensive products or services is useful. Practitioners can refer to the taxonomy systematically built with the accumulated knowledge of academia and other practitioners to extract what they need in specific domains.

We believe that our findings to collect and organise the different assets and terminologies used to describe the assets will help practitioners be more aware of each type of asset and how they are managed in the context.

3.5.4 Limitations and Threats to Validity

In this section, we cover the limitations of our work and how they might affect the results. The taxonomy is created based on the data extracted from the literature review, the field study, and academic expert knowledge. We combined the inputs from the mentioned sources to create the taxonomy. We designed the taxonomy to be extendable with new data identified by us and others in future studies as software engineering areas evolve (See Section 3.3.3). We encourage researchers and practitioners to consider the taxonomy within their organisation and identify the potential new types of assets and assets that can complement the asset management taxonomy's representativeness.

In the rest of this section, we cover the threats to construct, internal, and external validity as suggested by Runeson and Höst [180] and Runeson et al. [181].

Construct validity reflects the operational measurements and how the study represents what is being investigated. We are aware that the literature review is conducted in a limited area, i.e., TD. We chose the TD field as representative form of quality degradation since we believe TD is one form of asset quality degradation, as mentioned in Section 3.3.1. We acknowledge that the performed

snowballing and limiting the literature review to a specific topic might affect the construct validity of this work since we could not cover all the topics and the articles related to those topics. However this threat is mitigated with the additional data collection from the focus groups.

We acknowledge that the participants of the focus groups do not include all the relevant stakeholders in organisations. We tried mitigating this threat by involving participants with different roles and varying expertise from the companies. We are also aware that the participants' statements in the focus groups can be interpreted differently by the researcher and the participants. We mitigate this threat in three ways. First, by sending the summary, with the transcription of their statements and our own notes, back to the participants, asking for their feedback; second, by having two researchers code the raw data independently; and third, by choosing to code the data using the *in vivo* coding method, the qualitative analysis prioritises the participants' opinions.

External validity refers to the generalisability of the results and whether the results of a particular study can hold in other cases. We acknowledge and understand that the results are not comprehensive and might not be generalisable. The created taxonomy is based on the collected data and is extendable. We have provided a systematic way of extending the taxonomy, i.e., the meta-model. Finally, other threats that can affect the study's external validity are the number of involved companies, the country where the companies (investigated sites) are located, i.e., Sweden, and the involvement of all the roles in these organisations.

Reliability refers to the extent that the data and analysis are dependent on the researchers. When conducting qualitative studies, the goal is to provide results that are consistent with the collected data [147]. We have tried to mitigate this threat, i.e., consistency of the results. Firstly, by rigorously documenting and following the procedures of the focus groups to collect consistent data [225]. And secondly, by relying on consistency during the analysis, i.e., blind labelling of the data by multiple researchers and peer reviewing the labels.

3.6 Conclusions and Future Work

This paper presents a taxonomy for classifying assets with inherent value for an organisation subject to degradation. These assets are used during the development of software-intensive products or services. The creation of the taxonomy of assets attempts to provide an overarching perspective on various assets for

researchers and practitioners. The taxonomy allows us to characterise and organise the body of knowledge by providing a common vocabulary of and for assets.

Eight major types of assets are introduced in the taxonomy: assets related to *Product Requirements*, *Product Representation*, *Development*, *Verification and Validation*, *Operations*, *Environment and Infrastructure*, *Development Process/ Ways-of-Working*, and *Organisation*.

The taxonomy could be used for:

- Identify the gaps in research by providing the points of interest from practitioners' perspectives.
- Identify the state-of-the-art research for individual assets and their properties for practitioners.
- Communicate and spread the body of knowledge.

The taxonomy helps draw out the assets with similar characteristics and implicit relations among them. Most of such similarities of characteristics, properties, and relations are not immediately visible when considering the assets from only one perspective. Taking a more abstract and high-level look at the assets involved in the development of software-intensive products or services can help facilitate the management activities and the overall development process.

The dimensions provided by our taxonomy are not exhaustive, nor are the assets we identified. Therefore, we intend to conduct further investigation to complement the taxonomy by incorporating the new knowledge. Furthermore, we would like to study the relationship between assets, how the degradation of an asset can have a ripple effect and chain reactions, causing the degradation of other assets, and how the degradation impacts the development process. Lastly, we intend to investigate the individual properties of assets to identify the metrics used for measuring assets, their value, and their degradation (or lack thereof).

Besides, future and ongoing work will use the taxonomy as a base for further studies and exploration of assets, their characteristics, and the concepts of value, degradation and its different types.

Appendix: The Asset Management Taxonomy

Assets' definitions are presented in this section in Tables 3.8 to 3.16. The full tree of the asset management taxonomy is presented in Figure 3.13.

Table 3.8: Product-Requirements-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	**AM type**	**Definition**
Feature-Related Backlog Items	AM1.1	Feature-Related Backlog Items are the results of refining and breaking down the user stories to create executable tasks [151].
User Stories	AM1.1	User Stories are, according to the agile development paradigm, a way to specify the features of the software that is being developed [151].
Use Cases	AM1.1	Use Cases are lists of actions or events that describe how a user will achieve a goal in a system [105].
System Requirements	AM1.2	"System Requirements are the requirements for the system as a whole. System Requirements [...] encompass user requirements, requirements of other stakeholders (such as regulatory authorities), and requirements without an identifiable human source." [37].
User Interface Designs	AM1.2	"User Interface Design is an essential part of the software design process. User interface design should ensure that interaction between the human and the machine provides for effective operation and control of the machine. For software to achieve its full potential, the user interface should be designed to match the skills, experience, and expectations of its anticipated users." [37].
Quality Scenarios (The -ilities)	AM1.2	"A quality attribute (QA) is a measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders." [19] A quality scenario is a way of stating a requirement in an unambiguous and testable manner [19].
User Experience Requirements	AM1.2	User Experience Requirements "are considered key quality determinants of any product, system or service intended for human use, which in turn can be considered as product, system or service success or failure indicators and improve user loyalty." [109, 113].
Change Requests	AM1.3	Change Requests are the modifications to the software product that are not coming from the requirements analysis of the product.

Table 3.9: Product-Representation-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	**AM type**	**Definition**
Architectural Models	AM2.1.1	Architecture Models are partial abstractions of systems, they capture different properties of the system [108]. "Architecture modeling involves identifying the characteristics of the system and expressing it as models so that the system can be understood. Architecture models allow visualisation of information about the system represented by the model." [110]
Architectural Documentation	AM2.1.1	Architectural Documentation are the representations of the decisions made to construct the architecture of the software [108].
Design Decisions Documentation	AM2.1.2	Design Decisions Documentation are the results of the design decisions that architects create and document during the architectural design process [37].
System Designs	AM2.1.2	System Designs are the processes of defining elements of a system. These elements are specified in the requirements and are extracted to create modules, architecture, components and their interfaces and data for a system.
Documentation Automation Scripts	AM2.2	Documentation Automation Scripts are the scripts that generate documentation based on the state of the source code.
Product Documentation	AM2.2	Product Documentation are the operational guidelines (such as *user manuals* and *installation guides*) for when the product is in use.

Table 3.10: Development-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	**AM type**	**Definition**
Build Plans	AM3.1	Build Plans are the descriptions of how developers intend to build the software, i.e., by compilation of artefacts in a build chain, which will end in a running software.
Build Results	AM3.1	Build Results are the results of the build process, including the comments, documentation, and other artefacts that are generated during the build process.This is seen as a persistent asset if it holds more data than just an automated "throw away" report, and/or if the asset is used for reference over time.
Build Scripts	AM3.1	Build Scripts are the scripts that are used to run the build process.
Source Code	AM3.2	Source Code is the collection of code written in a human-readable and comprehensible manner stored as plain text [97].
Code Comments	AM3.2	Code Comments are the comments that developers integrate and write within the source code to clarify and describe certain parts of the code or its functionality [86].
APIs	AM3.2	APIs (Application Program Interfaces) are the interfaces that are created to facilitate interaction of different components and modules.
Architecture (Code Structure)	AM3.2	Architecture is the actual and fundamental relationships and structure of a software system and its source code [19].
Libraries/ External Libraries	AM3.2	Libraries/External Libraries are source code that belongs to the product but is not developed or maintained within the project, i.e., the developers. The software project depends on it and references the library.
Web Services	AM3.2	Web Services are running services on devices handling requests coming from networks
Versioning Comments	AM3.3	Versioning comments are the comments that developers submit to any version control application they use for the development. Such comments can later be extracted and viewed to identify the purpose of each event.

Table 3.11: Verification-and-Validation-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	**AM type**	**Definition**
Unit Tests	AM4.1	Unit Tests are the tests written to examine the individual units of the code [88]. "Unit tests generally focus on the program logic within a software component and on correct implementation of the component interface." [27]
Integration Tests	AM4.1	Integration Tests are the tests written to examine the combined set of modules as a group [27, 91].
System Tests	AM4.1	System Tests are the tests written to examine the system's compliance with the requirements.
Acceptance Tests	AM4.1	Acceptance Tests are the tests conducted to examine and determine whether the requirements are met according to the specifications of the requirements.
Non-Functional Test Cases	AM4.2	Non-Functional Test Cases are the tests that examine the quality of the system, i.e., non-functional aspects such as performance, availability, and scalability.
Test Plans	AM4.3	Test Plans are the documents that describe the testing scope and test activities that will be performed on the system throughout the development lifecycle.
Test Automation Scripts	AM4.4	Test Automation Scripts are the scripts that automate part of the testing process. More specifically, the scripts automate distinct testing activities or types of tests.
Test Automation (Real/ Synthetic) Data	AM4.4	Test Automation (Real/Synthetic) Data is the generated data that are used by the automation scripts to test the system.

Table 3.12: Operations-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	AM type	Definition
Customer Data	AM5	Customer Data is data that is collected from the customers (end users) of the software product such as user feedback.
Application Data	AM5	Application Data is the data that is created, collected, used, and maintained while developing the software product such as system performance.
Usage Data	AM5	Usage Data is the data that is collected while the software product is operational such as the data related to the performance of the system.

Table 3.13: Environment-and-Infrastructure-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	AM type	Definition
Deployment Infrastructure	AM6	Deployment Infrastructure are all the steps, activities, tools, process descriptions, and processes that facilitate the deployment of a software-intensive product.
Tools	AM6	Tools are any physical and virtual entities that are used for the development of a software product such as integrated development environments (IDE), version control systems, spreadsheets applications, compilers, and debuggers.
Tools Pipelines	AM6	Tool Pipelines are automated processes and activities that facilitate and enable developers to reliably and efficiently compile, build, and deploy the software-intensive product.

Table 3.14: Development-Process/Ways-of-Working-Related Assets are listed in this table. The table contains the definitions of the assets and their type. [The original table is divided into two tables because of page limitation – Tables 3.14 and Table 3.15]

Asset	**AM type**	**Definition**
Product Management Documentation	AM7.1	Product Management Documentation is any documentation that is used to facilitate the management activities and processes during the product development.
Documentation About Release Procedure	AM7.1	Documentation About Release Procedure is the description of the product release plan and the entities and activities associated with release.
Product Business Models	AM7.1	Product Business Models are the descriptions of how the organisation creates value for the customers with the software-intensive product.
Product Roadmap	AM7.1	Product Road Map is the abstract, high-level description of the evolution of the product during the development.
Product Scope	AM7.1	Product Scope is the description of the characteristics, functionality, and features of the software-intensive product.
Product Backlog	AM7.1	Product Backlog is any document that acts as a list where the features, change requests, bug fixes, and other similar activities are stored, listed, and prioritised.
Requirements Internal Standards	AM7.2	Requirements Internal Standards are the specific rules that the company introduces and utilises internally for dealing with the requirements of the product.
Architectural Internal Standards	AM7.2	Architectural Internal Standards are the specific rules that the development team introduces and utilises internally for designing, creating, and maintaining the architecture of the software-intensive product.
Documentation Internal Rules / Specifications	AM7.2	Documentation Internal Rules/Specifications are the specific rules that the development team introduces and utilises internally for creating and maintaining the documentation.

Table 3.15: Development-Process/Ways-of-Working-Related Assets are listed in this table. The table contains the definitions of the assets and their type. [The original table is divided into two tables because of page limitation – Tables 3.14 and Table 3.15]

Asset	**AM type**	**Definition**
Build Internal Standards	AM7.2	Build Internal Standards are the specific rules that the development team introduces and utilises internally for the build activities.
Coding Internal Standards/Specifications	AM7.2	Coding Internal Standards/Specifications are the rules that the development team introduces and utilises internally while developing the software-intensive product.
Versioning Internal Rules / Specifications	AM7.2	Versioning Internal Rules/Specifications are the rules that the development team introduces and utilises internally for version control during the development of software-intensive products.
Testing Internal Rules / Specifications / Plans / Strategies	AM7.2	Testing Internal Rules/Specifications/Plans/Strategies are the rules that the development team introduces and utilises internally for testing activities and procedures.
Process Internal Descriptions	AM7.2	Process Internal Descriptions are the descriptions of the procedures and activities that the development team introduce and utilise during the development of software-intensive products.
Process Data	AM7.2	Process Data is are the metrics and other information that concern the past and current status of the development process. Examples of such data are velocity, issues, bugs, backlog items, etc.
Documentation About Ways of Working	AM7.2	Documentation About Ways of Working are the description of work plans and working patterns, i.e., how the organisation and the development team plan to create and release the software-intensive product.

Table 3.16: Organisation-Related Assets are listed in this table. The table contains the definitions of the assets and their type.

Asset	AM type	Definition
Organisation's Structure	AM8	Organisation's Structure is the description of how the organisation directs the activities to achieve organisational goals.
Organisation's Strategy	AM8	Organisation's Strategy is the description of the plans that guide the organisation how to allocate its resources to support the development of the software-intensive product.
Business Models	AM8	Business Models are the descriptions of how the organisation creates value with the software-intensive product for the organisation.

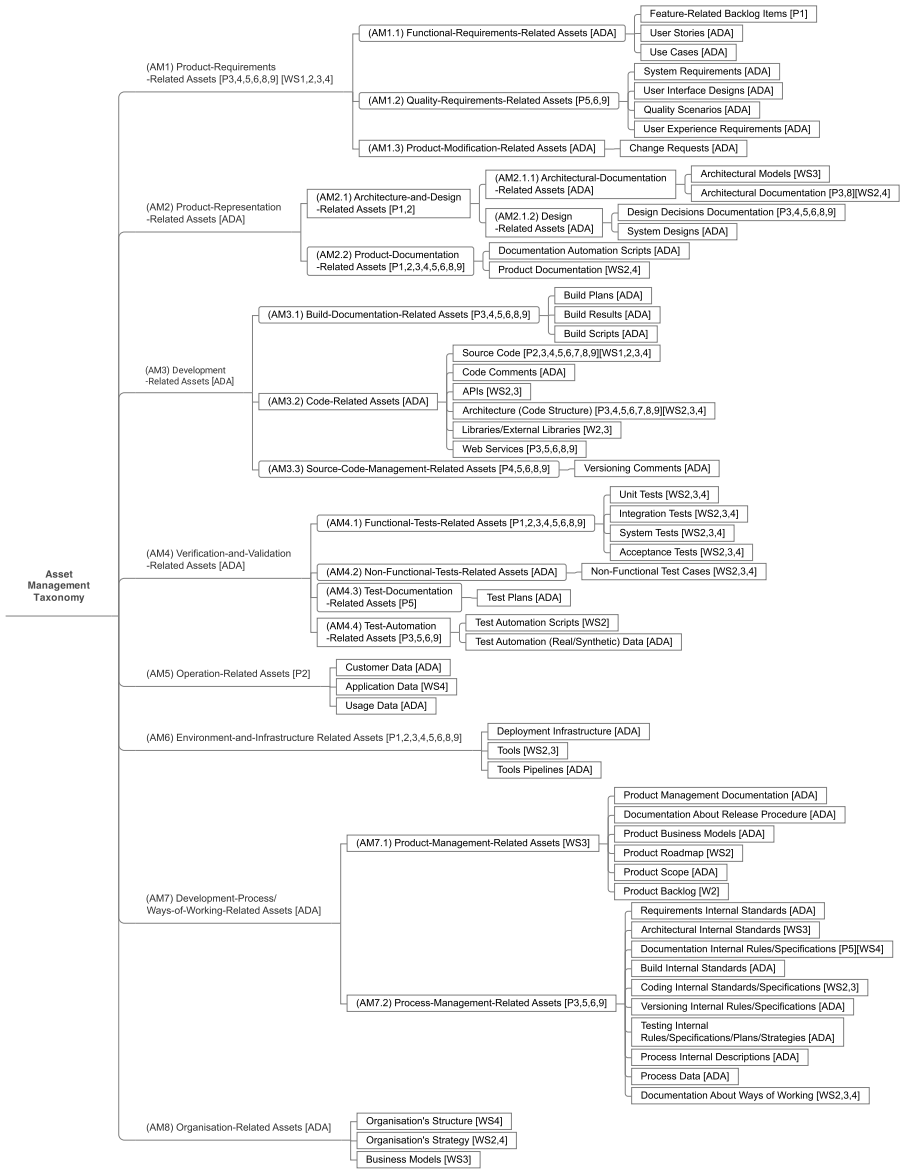

Figure 3.13: The asset management taxonomy.

Chapter 4

Refactoring, bug fixing, and new development effect on technical debt: An industrial case study

This chapter is based on the following paper:

> **Ehsan Zabardast**, Javier Gonzalez-Huerta, and Darja Šmite. "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study" *In 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 376-384). IEEE (2020).[1][2][3]

[1] *© 2020 IEEE. Reprinted, with permission, from Ehsan Zabardast, Javier Gonzalez-Huerta, and Darja Šmite "Refactoring, Bug Fixing, and New Development Effect on Technical Debt: An Industrial Case Study", 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2020*

[2] https://doi.org/10.1109/SEAA51224.2020.00068

[3] *In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of BTHs products or services. Internal or personal use of this material is permitted.*

Code evolution, whether related to the development of new features, bug fixing, or refactoring, inevitably changes the quality of the code. One particular type of such change is the accumulation of Technical Debt (TD) resulting from sub-optimal design decisions. Traditionally, refactoring is one of the means that has been acknowledged to help to keep TD under control. Developers refactor their code to improve its maintainability and to repay TD (e.g., by removing existing code smells and anti-patterns in the source code). While the accumulation of the TD and the effect of refactoring on TD have been studied before, there is a lack of empirical evidence from industrial projects on how the different types of code changes affect the TD and whether specific refactoring operations are more effective for repaying TD. To fill this gap, we conducted an empirical study on an industrial project and investigated how Refactoring, Bug Fixing, and New Development affect the TD. We have analyzed 2,286 commits in total to identify which activities reduced, kept the same, or even increased the TD, further delving into specific refactoring operations to assess their impact. Our results suggest that TD in the studied project is mainly introduced in the development of new features (estimated in 72.8 hours). Counterintuitively, from the commits tagged as refactoring, only 22.90% repay TD (estimated to repay 8.30 hours of the TD). Moreover, while some types of refactoring operations (e.g., Extract Method), help repaying TD, other refactoring operations (e.g., Move Class) are highly prone to introduce more TD.

Keywords: *Technical Debt, Empirical Study, Industrial Study, Case Study, Refactoring, Bug Fixing, New Development*

4.1 Introduction

Technical Debt (TD) is a metaphor used to discuss the long-term consequence of sub-optimal design decisions taken when short-term goals are prioritized [55], and motivate the importance of refactoring for primarily nontechnical stakeholders [107]. TD inevitably accumulates and evolves as the software develops, and as it is maintained [178]. TD is infamous for its negative impacts on software maintainability and evolvability [14, 106]; and has, therefore, become an important research topic in modern software engineering.

Technical Debt has been approached by researchers from many different angles. On the one hand, one research direction in the area of technical debt has focused on the "rhetorical discussions" about the use of the metaphor [107], ways for identifying and measuring the debt, causes, and effects of the technical debt.

On the other hand, several other research works focus on the understanding of how to deal with the TD, in particular, TD repayment, with refactoring being one of the most common topics [128].

Refactoring is "the process of changing a software system in such a way that does not alter the external behavior of the code yet improves its internal structure" [76]. Certain refactoring operations aim at improving the maintainability of the code, while others are aiming at improving the understandability or having more "clean code" (as defined in [134]), or remediating the TD by removing a TD item [8, 106, 200]. TD items (TDI) are "single elements of TD," something that can be identified in the code [106], and have been introduced to be able to quantify or visualize the TD. In their turn, refactoring operations are presented in refactoring catalogs, such as the ones presented by Fowler [76, 77]. The catalogs provide the motivation behind each refactoring operation and the circumstances in which it should be used.

Yet, refactoring is not the only way to address the TD. As software evolves, developers perform various manipulations to the code that can be categorized into three major types of activities: refactoring, bug fixing, and new feature development, all of which are likely to have some impact on the TD. Related studies show, for example, that developers spend, on average, 25% of the development time on managing the TD [135], while these activities are not always performed systematically but rather sporadically during the development process [31, 135]. Similarly, Palomba et al. [166] and Kim et al. [99] found that refactoring operations are mostly performed when new features are implemented and not as a result of dedicated code maintenance. At the same time, any manipulation of the code, whether related to the development of new features, bug fixing, or refactoring, inevitably changes the quality of the code and often results in the accumulation of TD. However, to the best of our knowledge, the research comparing the evolution of TD linked to the different types of activities (refactoring, bug fixing, and new development) is scarce.

In this article, we investigate the effects that refactoring, bug fixing, and new development have on the accumulation of the TD. In addition, we further delve into specific refactoring operations to understand how these refactoring operations, that are expected to improve the internal quality of the source code [76, 77], impact on the accumulation of repayment of TD. Therefore, we aim at answering the following research questions:

- **RQ1.** *To what extent do activities marked as Refactoring, Bug Fixing, and New Development affect the accumulation of Technical Debt in the project?*

- **RQ2.** *How each specific type of refactoring operations affects the Technical Debt in the project?*

This is done by conducting an empirical study where we analyze a large-scale industrial project commit by commit, to assess the impact that each commit has on the accumulated TD. In this project, developers systematically tagged their commits to identify the activity addressed in the commit. We used two independent tools to i) calculate TD and ii) detect the refactoring operations in each commit. We merge the results to evaluate how TD is affected by the activities. We analyzed 2,286 commits in total and investigated whether the total TD was reduced, remained the same as before, or increased for each activity, and later by the specific refactoring operation.

The rest of this article is structured as follows. Section 4.2 summarizes the related work. Section 4.3, research methodology, describes how the data was collected and analyzed. The results are presented in Section 4.4, and the implications of the results are discussed in Section 4.5. Section 4.6 discusses the threats to validity. Lastly, Section 4.7 concludes the paper.

4.2 Related Work

Managing TD is essential, and companies have different approaches on how to address this problem. In particular, tracking TD and how it impacts the development is of particular interest for industry and academia. In [135], the authors investigate the state-of-practice in managing TD and aim to understand how companies track TD, what tools they use, and what is the cost of managing TD. They found that, on average, 25% of development time is spent on managing TD. In [31], the authors investigate the waste of development time with regards to TD management. Their results suggest that developers waste 23% of their time on managing TD (i.e., mainly through refactoring), and developers frequently introduce new TD.

Refactoring the code is one of the strategies to deal with TD, and it has been investigated before [22, 23, 60, 119, 165, 185, 189, 200, 214, 227]. In [23], the authors investigate the relationship between code quality and refactoring operations. They conclude that there is no clear relationship between the refactoring operations and code quality because the refactoring operations mostly target the code components that quality metrics do not consider as in-need-of-improvement. Palomba et al. [165] investigate the perception of developers on code smells. They summarize their findings in four lessons: not all the code

smells are considered as design flaws; the "intensity" of the problem is an indication of it being a code smell or not; the complex or long source code are generally an important sign of code smells; and, the experience of developers is a key when identifying a CS.

The impact of refactoring operations, in general, have mostly been studied on code smells, which is only one type of TDI. Santos et al. in [185] vestigate the impact of code smells on software development. In [79], Fujiwara et al. propose a method to assess the benefits of refactoring instances in maintainability. They use three metrics; namely *refactoring frequency*, *defect density*, and *fix frequency*. They conclude that after a term with higher refactoring frequency, defect introduction decreases. In their paper, Tufano et al. [214] studied "when the code smells are introduced" and "what is the survivability of the code smells." They conducted a study over the change history of 200 open source projects from Apache, Android, and Eclipse ecosystems. They focused on five different code smell types, namely *Blob Class*, *Class Data Should be Private*, *Complex Class*, *Functional Decomposition*, and *Spaghetti Code*. They concluded that "most of the smell instances are introduced when an artifact is created and not as a result of its evolution." They also found that 80% of the code smells remain in the system. Finally, from the remaining 20% of the code smells removed, refactoring operations only remove 9% of the code smells. In a similar study, Yoshida et al. [227] analyzed the refactoring data and code smells detected in Apache Ant, ArgoUML, and Xerces-J. They investigate the effectiveness of refactoring patterns applied to code smells to answer whether the refactoring operation helped to remove the code smells. The results of their investigation concluded that "... refactoring rarely removes the code smell because the corresponding pattern is rarely applied to a code smell." In a similar article, Palomba et al. [166] investigate the relationship between refactoring operations and code changes. Their results indicate that most of the refactoring operations are done to remove the duplicated code or "previously introduced self-admitted technical debt."

The articles that study TD management approaches have an overview of what activities impact TD [128]. Digkas et al. [60] studied fifty-seven open-source projects to investigate how TD accumulates during the maintenance process. They found out that a small proportion of issues (types of TDIs in SonarQube terminology) are responsible for the repayment of the large percentage of TD. They also studied the evolution of TD, considering other activities than refactorings. Silva et al. [189], similar to the previous study, investigated the motivation behind applying specific refactoring operations. They concluded

that refactoring operations are mostly performed when new requirements are presented rather than to remove existing code smells.

Our study aims to fill in the gaps that are not covered in previous studies by combining their strengths and different perspectives. While previous studies mostly focus on specific type of activities (primarily refactoring) and their impact on specific types of TDI (code smells), our study compares the impact of different activities (refactoring, bug fixing, and new development) and their impact on TD. In contrast to many studies of open source projects and sole reliance on automatically detected refactoring operations, we have a better certainty over the nature of activities by using the tags that developers systematically introduced in their commits, and thus their intention. Finally, while previous studies use open-source data, we use industrial data to have insights from the state-of-the-practice.

4.3 Research Methodology

To address the research questions, we designed an empirical study where we analyzed data gathered through archival analysis. We selected a large-scale (approximately 1.5 million LOC) industrial project from a company that chose to stay anonymous. The product provides financial services via mobile phones and the internet (FinTech global product). The software in the project was written in Java and has evolved for more than ten years. The project has followed core Agile practices (e.g., continuous integration) with frequent releases. We chose this project based on convenience (availability and access) and because we could extract developers' intention for each specific commit activity through the systematically documented commit tags. The analysis is circumscribed to a period of one year, where the project was under heavy development process. We expected to have more activity on the code during this period, which encompasses more development and refactoring. It is important to highlight that this project was developed with some specific development practices such as "clean-code", test-focused development, high emphasis on refactoring, and having a reliable regression test suite in place.

In the following, we describe the main constructs and measurements used in our study (summarized in Figure 4.1). We analyzed i) the type of commit activity based on the tag provided by developers for each commit; ii) the amount of TD in a given commit; iii) refactoring operations in each commit. We used two tools to collect the data: i) SonarQube[4] for calculating the TD in the project,

[4]https://www.sonarqube.org (version 6.7.4)

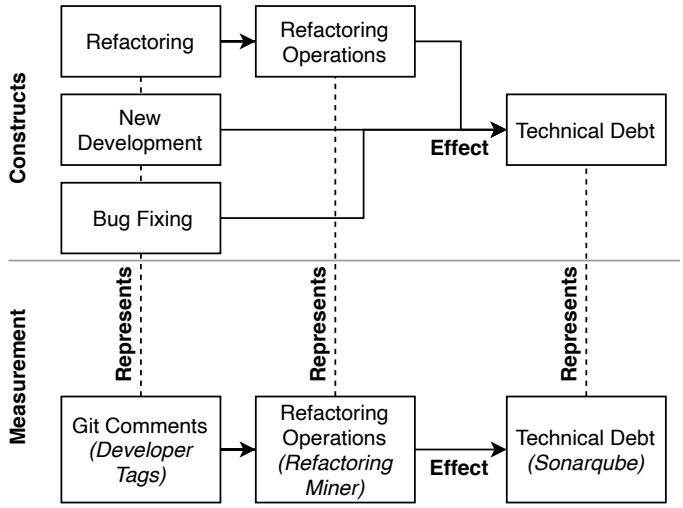

Figure 4.1: The Study Constructs and Measurements.

and ii) RefactoringMiner[5] to detect the refactoring operations (ROs). Both tools have been previously used in similar studies of refactorings [22] and TD [60]. The data was collected by the tools separately in two steps and then combined for the analysis using the git hash identifier for each commit. The details of how the data was collected in each step are described further in Section 4.3.1.

4.3.1 Data Collection

Detecting Types of Code Change Activities

As described in the case description, the comment field in each commit was systematically tagged by the developers, which helped us determine the intention of that particular commit. We classified the activities on the commits in six categories, namely *Bug Fixing*, *Build*, *Commit Merge*, *New Development*, *Refactoring*, and *Other*.

We discarded the commits tagged with *Build* and *Other* since we are interested in commits that i) we can extract developers' intent; and ii) are related

[5]https://github.com/tsantalis/RefactoringMiner

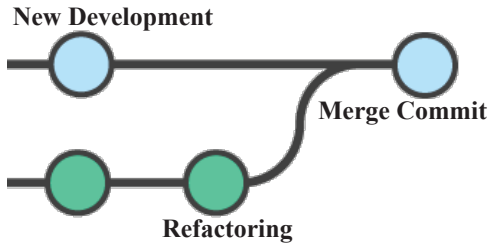

Figure 4.2: Merge Commit Investigation Example.

to the activities in the scope of this paper (i.e., refactoring, bug fixing, and new development). In the case of *Merge Commits*, we looked into the branches before the merge commits to investigate them instead. As an example, in Figure 4.2, instead of investigating TD in the *Merge Commit*, we looked back at the commits of the two branches, i.e., *Refactoring* and *New Development*, to analyze the activities and TD in those commits. The details of the collected data are presented in Table 4.1.

Table 4.1: Demographics of The Activities.

Name	**# of Instances**	**Percentage**
Bug Fixing	226	9.88%
Build	199	8.70%
Commit Merge	578	25.28%
New Development	801	35.03%
Refactoring	476	20.82%
Other	6	0.26%
Total	2286	100%

Detecting Technical Debt

SonarQube is an open-source tool used for code quality inspection for a software project, which analyzes the source code in order to detect bugs, code smells, and security vulnerabilities (what in SonarQube terminology is referred to as issues). Additionally, SonarQube provides the effort in time, which is calculated based

on the remediation effort function. Therefore, the technical debt of a project is the summation of the estimated time needed to solve all the issues. We used the default profile in SonarQube for calculating the remediation time.

Detecting Refactoring and Refactoring Operations

We used RefactoringMiner to extract the detected refactoring operations performed on commits classified as refactoring. RefactoringMiner is a library developed by Tsantalis et al. [211] that detects various types of refactoring operations in the history of a JAVA project. The latest version of RefactoringMiner can detect 40 different types of ROs with the precision of 98% and recall of 87% [211]. We use RefactoringMiner to first detect the ROs in the history of the project in all the investigated commits. Later, we filtered and investigated only the ROs that are tagged as a "refactoring" commit by the developers.

4.3.2 Data Analysis

To analyze the effect of Refactoring, Bug Fixing, and New Development, we calculated $\Delta TD_j = \sum e_j - \sum e_{j-1}$ (e: effort in minutes)—i.e., the TD introduced or removed by commit *j*. ΔTD_j is the difference in TD between commit *j*, in which the activity has happened, and the TD of the previous commit TD (i.e., commit *j-1*). The sign of ΔTD_j determines how TD is affected. If the sign is positive, it means that the project accumulates more TD with that particular commit (i.e., TD increases). If the sign is negative, it means that the TD was paid back with that particular commit (i.e., TD decreases). If ΔTD_j is *zero*, it means that TD has not changed with that particular commit.

To analyze how individual Refactoring Operations affect TD, we use the same concept of ΔTD_j. However, we analyze the data with a different granularity level.

We utilize the data collected by the RefactoringMiner tool to extract the ROs that happened on specific files of the commits tagged as refactoring. The developer expressed that the primary purpose of the commit was performing refactoring. We look at the issues (if any) that were introduced or removed by that particular commit in the files in which RefactoringMiner detected ROs. With this data, we can trace whether particular ROs happening in a file (or pairs of files) have an effect on ΔTD_j for those particular files (or pairs of files). Figure 4.3 illustrates an example of how ΔTD_j is calculated when an RO removes an issue.

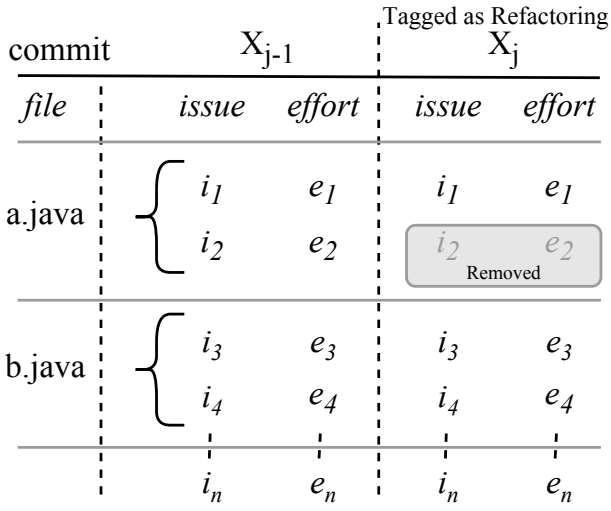

Figure 4.3: An Example of How ΔTD is Calculated Where an Issue is Removed.

4.4 Results

In this section, we present our results from studying the impact of refactoring (also focusing on specific refactoring operations), bug fixing, and new development on Technical Debt. Table 4.2 summarizes the descriptive statistics related to the three types of activities and their impact on the accumulation and repayment of TD. As we can see, Refactoring and Bug Fixing have on average negative impact on TD (overall mean of -1.04 and -0.2 minutes respectively), meaning that refactoring and bug fixing, at large, helps in repaying TD. The mean for commits tagged as New Development is 5.45 minutes meaning that, overall, it contributes to the accumulation of TD.

Figure 4.4 illustrates the accumulation of TD in hours in the project during the period under analysis (i.e., prior to the removal of commits tagged as *Build*, *Commit Merge*, and *Other*). Commit 1 in this picture represents the initial commit on the time-span of our analysis we investigated. The total amount of TD increased by $67.72h$ during the analyzed time-span, although, as we can observe in Figure 4.4, it fluctuates significantly during that period. Our results further suggest that, in most cases, TD is introduced during the *New*

Table 4.2: Statistics for the Collected Data on Refactoring (R), Bug Fixing (BF), and New Development (ND).

	N	Mean	SD	Effect on TD	Cases Adding TD	Cases Repaying TD
R	476	$-1.04'$	$29.30'$	$-8.3h$	106	109
BF	226	$-0.2'$	$20.89'$	$-0.77h$	40	44
ND	801	$5.45'$	$38.87'$	$72.8h$	250	158

Development, while *Refactoring* and *Bug Fixing* were found to, on average, remove TD. There are four major changes in TD highlighted in Figure 4.4 with red vertical solid lines for major increases and green vertical dashed lines for major decreases:

1: A sharp increase of TD tagged as new development.

2: A significant decrease of TD tagged as refactoring (tagged as *"Removed legacy value"*, *"Clean up"*, and *"Removed dead code"*).

3: A sharp increase of TD tagged as new development.

4: A significant decrease of TD tagged as refactoring and bug fixing (tagged as *"Fixing failing test cases"*).

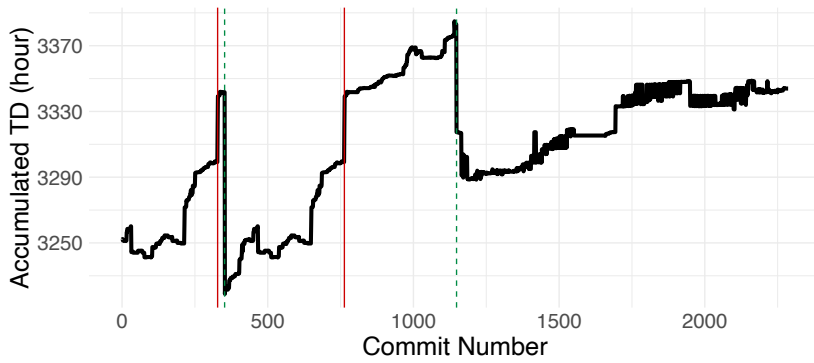

Figure 4.4: The Evolution of Accumulated Technical Debt (hours) in the Project.

Figure 4.5 illustrates the impact of each activity on the accumulation of TD in the project. The commits illustrated in this figure are sequential but not consecutive because, as described in Section 4.7, we have removed the commits tagged as *Build*, *Commit Merge*, and *Other*. The red bars in the figure show the total amount of the TD introduced by a given commit, whereas the green bars show the total amount of the TD removed in a given commit.

As it could have been expected, our results indicate that *New Development* is the primary source of accumulation of TD. The 801 commits tagged as *New Development* contributed to the accumulation of TD by $72.8h$ hours in total. As shown in Figure 4.6, 31.21% (250 cases) of the commits marked as *New Development* contributes to the accumulation of TD, while 19.73% (158 cases) contributed to its repayment. This can be owing to the fact that the commits tagged as the development of new features might also include refactorings to introduce design or architectural changes. Previous research suggests that refactoring most likely occurs during new development or bug fixing (e.g., [99, 166]).

In the analyzed period, the commits tagged as *Bug Fixing* contributed to the repayment of TD. The 226 commits tagged as bug fixing contributed to the repayment of TD by $0.77h$ (i.e., 46 minutes) in total. While 16.54% (44 cases) of the commits repaid TD, 15.03% (40 cases) contributed to the accumulation of TD, and 62.83% had no impact on the TD in the project.

Refactoring is the main activity that contributes to the repayment of TD. The 476 commits tagged as refactoring contributed to the repayment of TD by 8.30 hours in total. While 22.90% (109 cases) of the commits repaid TD, 22.27% (106 cases) contributed to the accumulation the TD, and 54.83% had no impact on the TD in the project.

To further detect the individual Refactoring Operations that had happened in the commits tagged by the developers as "Refactoring," we used RefactoringMiner tool. Our analysis of the specific ROs suggests that out of 40 different types of ROs that the tool detects, there are only 22 types present within the analyzed commits. The impact of these ROs on the TD is illustrated in Figure 4.7.

Out of 22 ROs detected, 5 ROs, namely *Replace Attribute*, *Pull Up Method*, *Parametrize Variable*, *Move Attribute*, and *Extract Superclass* helped to remove the TD in all the cases. Three ROs, namely *Push Down Method*, *Pull Up Attribute*, and *Extract Subclass* were found to have no effect on the TD, while *Extract and Move Method* were found to accumulate TD in all cases. *Replace Attribute*, *Rename Variable*, *Rename Parameter*, *Rename Method*, *Rename Class*, *Rename Attribute*, *Pull Up Method*, *Extract Variable*, and *Extract Method* can be said to be the most effective in removing TD. At the same time, *Rename*

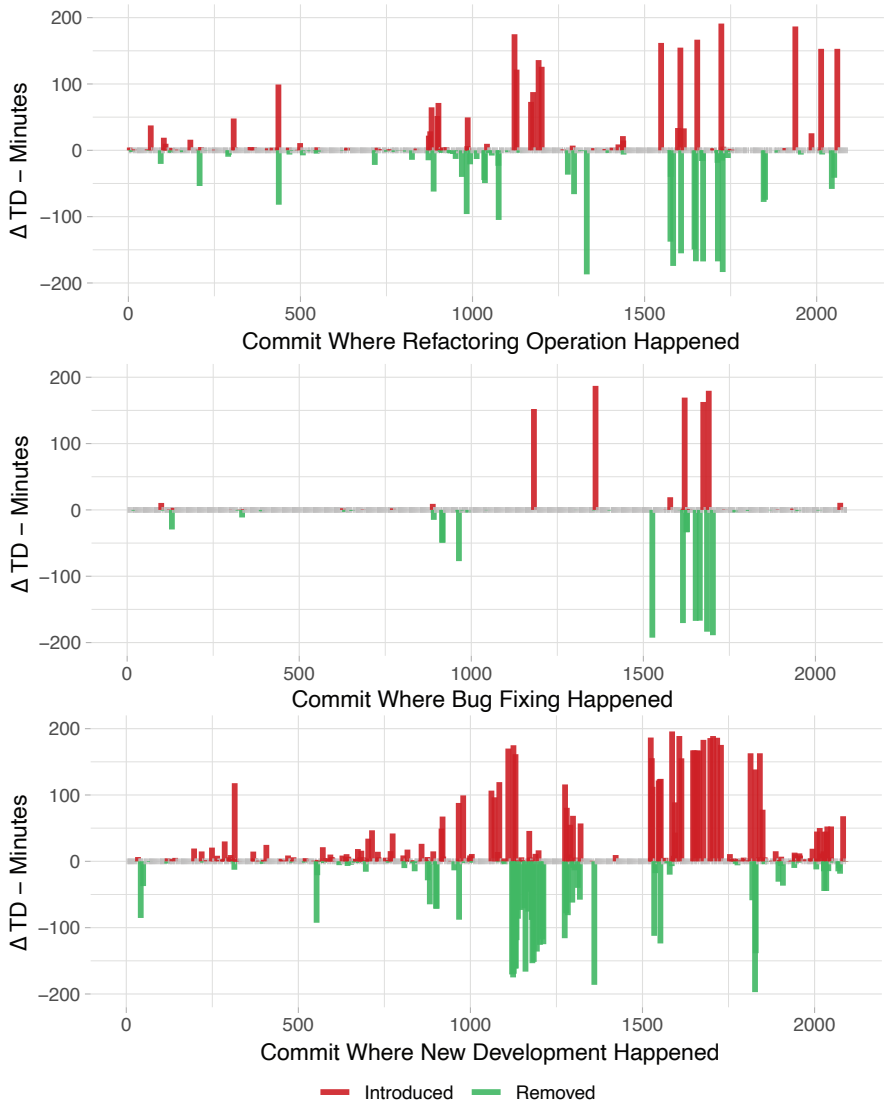

Figure 4.5: The Illustration of the Impact of Activities on ΔTD During the Evolution of the Code.

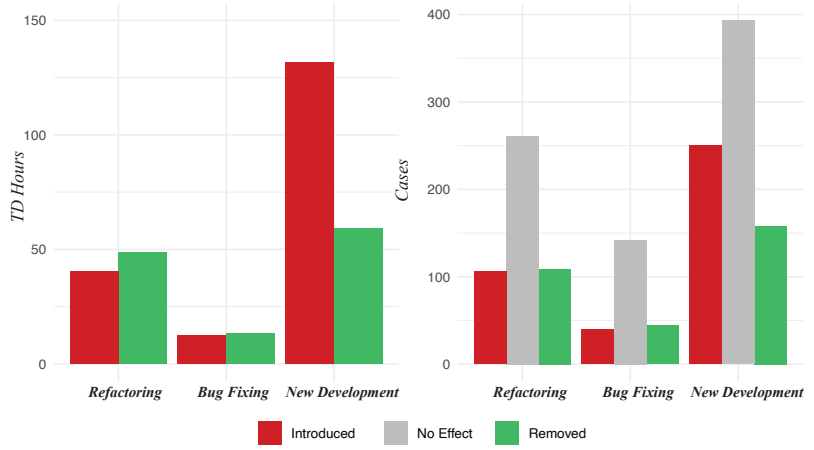

Figure 4.6: The Total Number of Cases and TD, in hours, each Activity Introduces or Removes from the Project.

Class, *Move Class*, *Move and Rename Class*, *Extract and Move Method*, and *Change Package* contribute to the accumulation of TD, although in some cases can also help repaying TD. Figure 4.7 (on the left side) together with Table 4.3–4.4 illustrate and summarize the ratio of observations for the RO types.

4.5 Discussion

In this section, we discuss our results with regards to our research questions, followed by the implications of the major findings for future research and practice.

4.5.1 RQ1: The effect of Refactoring, Bug Fixing, and New Development on TD

As already highlighted in Section 4.4, Refactoring, Bug Fixing, and New Development affect the accumulation of TD in very different ways. We have observed that commits tagged as *Refactoring* are, in general, contributing to the repayment of TD, but also that some commits tagged as *New Development* are contributing to its repayment. However, in the majority of the cases commits

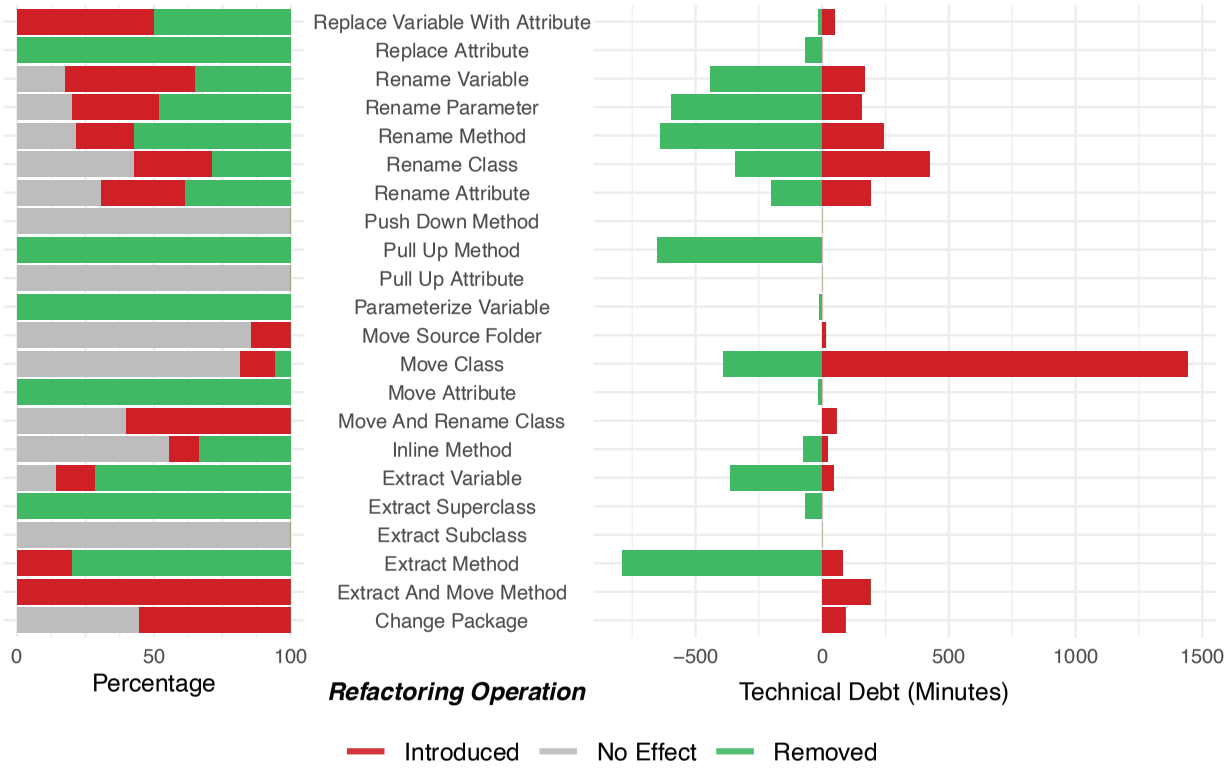

Figure 4.7: Refactoring Operations Types Detected within the Analyzed Commits in the Project.

Table 4.3: Refactoring Operations Summary of Results - The Table Summarizes the Total Number of Detected Cases, the Division of *Removed*, *Introduced*, and *No Changes* Cases with Their Respective Ratios, and the Amount of Introduced and Removed TD in Minutes. [The original table is divided into two tables because of page limitation – Tables 4.3 and Table 4.4]

	Refactoring Operation	**# Total**	**# Removed**	**TD Minutes Removed**	**# Introduced**	**TD Minutes Introduced**
1	Replace Variable with Attribute	2	1	-14	1	49
2	Replace Attribute	1	1	-65	0	0
3	Rename Variable	23	8	-443	11	168
4	Rename Parameter	25	12	-596	8	156
5	Rename Method	28	16	-640	6	243
6	Rename Class	21	6	-343	6	424
7	Rename Attribute	13	5	-201	4	192
8	Push Down Method	3	0	0	0	0
9	Pull Up Method	10	10	- 650	0	0
10	Pull Up Attribute	2	0	0	0	0
11	Parametrize Variable	1	1	-10	0	0
12	Move Source Folder	7	0	0	1	15
13	Move Class	682	39	-390	86	1442
14	Move Attribute	1	1	-14	0	0
15	Move and Rename Class	5	0	0	3	57
16	Inline Method	9	3	-76	1	22
17	Extract Variable	7	5	-363	1	45
18	Extract Superclass	1	1	-65	0	0
19	Extract Subclass	1	0	0	0	0
20	Extract Method	15	12	-791	3	82
21	Extract and Move Method	1	0	0	1	190
22	Change Package	9	0	0	5	91
	Total	**867**	**121**	**-4661**	**137**	**3176**

Table 4.4: Refactoring Operations Summary of Results - The Table Summarizes the Total Number of Detected Cases, the Division of *Removed*, *Introduced*, and *No Changes* Cases with Their Respective Ratios, and the Amount of Introduced and Removed TD in Minutes. [The original table is divided into two tables because of page limitation – Tables 4.3 and Table 4.4]

	Refactoring Operation	**# Total**	**# No Change**	**% Removed**	**% Introduced**	**% No Change**
1	Replace Variable with Attribute	2	0	50	50	1
2	Replace Attribute	1	0	100	0	0
3	Rename Variable	23	4	34.78	47.83	17.39
4	Rename Parameter	25	5	48	32	20
5	Rename Method	28	6	57.14	21.43	21.43
6	Rename Class	21	9	28.57	28.57	42.86
7	Rename Attribute	13	4	38.46	30.77	30.77
8	Push Down Method	3	3	0	0	100
9	Pull Up Method	10	0	100	0	0
10	Pull Up Attribute	2	2	0	0	100
11	Parametrize Variable	1	0	100	0	0
12	Move Source Folder	7	6	0	14.29	85.71
13	Move Class	682	557	5.72	12.61	81.67
14	Move Attribute	1	0	100	0	0
15	Move and Rename Class	5	2	0	60	40
16	Inline Method	9	5	33.33	11.11	55.56
17	Extract Variable	7	1	71.43	14.29	14.29
18	Extract Superclass	1	0	100	0	0
19	Extract Subclass	1	1	0	0	100
20	Extract Method	15	0	80	20	0
21	Extract and Move Method	1	0	0	100	0
22	Change Package	9	4	0	55.56	44.44
	Total	**867**	**609**	**-**	**-**	**-**

have no global impact on the accumulated TD (i.e., grey bars in Figure 4.6). This might be owing to the fact that, although the product under study has strict rules to tag commits, developers can be amalgamating changes in a single commit. For example a commit tagged as *New Development* might also contain many refactoring operations to prepare the architecture for the new code added, and even some bug fixes, as suggested in previous research in the area [99]. Refactorings can also remove TD on a given code entity but at the same time introduce the same amount of TD elsewhere, and the total might have 0 balance (as if there has been no effect). In addition, sometimes refactoring operations might contribute to improving the quality of the code (by for example reducing the size of a class without achieving the required length to remove the Large Class code smell).

When considering the whole project, it might seem that refactoring might not have a big impact on TD, but as illustrated in Figure 4.4, without the major refactoring events that usually happen after new development, the accumulated TD will grow very quickly and out of control.

Commits tagged as New Development contribute to the accumulation of TD in 31.21% of the cases. However, New Development also helps repaying TD in 19.73% of the cases. Refactoring, on the other hand, is expected to have substantial effect on the TD accumulated in the project. However Refactoring is only responsible for the removal of TD in 22.90% of the cases; in 54.83% cases, do not change the amount of TD; and, counterintuitively, it introduces additional TD in 22.27% of the cases. While Refactoring is more effective in removing TD than Bug Fixing and New Development, given its purpose, it's not living up to the expectations. Refactoring, by definition, is performed to increase the code quality. Even though Refactoring slightly out performs the other activities in removing TD, it still introduces substantial amount of TD in a high proportion of cases, and in half of the cases, does not introduce observable changes in the overall amount of TD accumulated in the project. These results are similar to the previous studies (e.g., [22, 23, 31]) that suggest that developers might waste a significant proportion of their time dealing with TD in an inefficient way [31] and that refactoring operations were not found to be effective in removing TD items as Code Smells [23], or even were responsible for introducing bugs [22].

Bug Fixing activities, by definition, are not aimed to deal with TD. Bugs are not Technical Debt Items *per-se*; however, they can be the consequence of TD. The TD removed by bug fixing activities might be accidental, e.g., due to the deletion of parts of the code, or refactorings being embedded in a bug-fixing-tagged commit.

This overall lack of effectiveness of refactoring to repay TD might be owing to the fact that developers do not refactor the code only to remove TD, but also to make design or architectural changes to enable other modification in the code base. Other explanations might be: i) certain refactoring operations are not specifically designed to mitigate TD, and ii) that certain operation can have non-trivial side effects that introduce TD in a bigger amount than the TD that helps removing. And lastly, not all the TD can be resolved by simply refactoring the code.

4.5.2 RQ2: The effect of Refactoring Operations on TD

In response to our second research question, we observe that some types of Refactoring Operations are more effective in removing TD such as *Extract Method* and *Pull Up Method*. The results suggest that the refactoring operations which has to do with more than one file, i.e., changing more than one file in the same refactoring (e.g., *Extract and Move Method*, *Move Class*, and *Move and Rename Class*) tend to increase the total amount of TD. Our results are aligned with the findings of [22] that suggest the refactoring operations involving hierarchies are prone to introduce faults. Therefore these refactoring operations should be used cautiously with more accurate code inspection and testing activities.

4.5.3 Implications for Research and Practice

- **Implications for researchers**: There is still room for further research in this area. The findings of this study can be used as a guideline to further investigate where the refactoring operations should be used to be more effective, i.e. whether the refactorings are happening in the hotspots (the file with frequent changes) follow the same pattern. Further analysis is required to investigate whether there exists a correlation between the total number of refactorings and the total amount of Technical Debt. These results can lead to a better understanding of how to utilize refactorings and thus improve TD management.

- **Implications for practitioners**: As mentioned before, some specific types of refactoring operations seem to yield better results when applied to the code while others seem to exacerbate the code quality. Practitioners can utilize these results when prioritizing the refactoring operations they use while maintaining the code. Lastly, the refactoring types that contribute to increasing the total amount of TD should be used cautiously.

Generally, the mapping between theoretical constructs and their representation are essential when building tools and maybe of interest to raise as a challenge and focus of the projects when building decision support tools.

4.6 Threats to Validity

The results of this study are subject to threats to *construct validity*, *internal validity*, and *external validity*.

Construct validity refers to the relationship between the theory and the measurements of the observations. It is the most critical threat for this study which concerns the collection of the data, and it is related to the limited scope of the analysis. We only analyze a limited number of commits. These commits are tagged by developers to identify the type of activities performed on that commit. We rely on a tool to detect refactoring operations. More specifically, the threats to the construct validity of the study are:

- *Imprecise identification of the refactoring operations*: We only studied the refactoring operations that have happened in the commits tagged by the developers as *Refactoring*. Further, to detect the individual refactoring operations in refactored commits, we have used RefactoringMiner. We have tried to mitigate this threat by taking the developers' intent into account and by selecting a state-of-the-art tool whose accuracy has been analyzed.

- *Imprecise calculation of Technical Debt*: We have tried to mitigate this threat by using SonarQube which is broadly used tool measuring TD and has been also employed in similar research studies (e.g., [60, 182]). We have used the default remediation that SonarQube associates to TD items, since we believe the results can be more repeatable, and also because practitioners might be reluctant to customize static code analysis tools [217].

- *Developers incurring in unintentional TD*: TD per definition refers to taking the shortcut intentionally, but this is the most rare case (as discussed in [106] pp. 153-154). We cannot be certain about whether the TD was introduced intentionally, and we only rely on the results provided by SonarQube. This might threaten our results, and we will investigate this issue in future empirical studies.

- *TD being introduced not in the files affected by refactoring operations*: When analyzing RQ2 we have minimized this threat by circumscribing the analysis to the files affected by each refactoring operation.

Threats to internal validity refer to confounding factors that might affect the results. The first threat to the internal validity comes from the association between the main activity tagged by developers in the commit and the different activities that can contain in reality. We make the analysis relying on the information tagged by the developer, but a given commit might of course comprise several different activities. However we are analyzing the main *intent* expressed by the developer. In the case of a refactoring commit, it can contain not only refactoring operations, but the goal of the commit is improving the quality of the code, therefore one can expect a positive impact on the global TD of the project. Our results might have been affected by this fact and we plan to deeper investigate this phenomena in further empirical studies.

Threats to external validity refer the generalizability of the results. In this study, we have analyzed a large scale industrial project which is mainly developed in Java. We understand that the generalizability of the results is limited, and we can only claim that our results are applicable to the analyzed context. We plan to replicate this study not only in other industrial but also in Open-Source projects.

4.7 Conclusion

In this paper we present an empirical study for investigating the impact of refactoring, bug fixing, and new development on technical debt, further delving into specific refactoring operations to assess their impact. We have analyzed 2,286 commits from a large scale industrial project, commit by commit in file level. Our results, within the studied project, show that overall Refactoring help mitigating TD. However we have also found that the majority of the cases Refactoring has no effect on TD, and it can even contributes to the accumulation of TD, which is in line with previous results (e.g., [23]). The TD is mainly introduced during the development of new features, although we have also observed that commits tagged as new development can help repaying TD, which might be owing to the fact that commits tagged as new development might contain refactoring operations embedded, as found in previous research [99]. Finally, Bug fixing was found to contribute to the repayment of TD in the studied period.

Further, we have investigated the impact of specific refactoring operations (ROs) on TD. The results suggest that some ROs namely Extract Method, Pull

Up Method, Rename Method, Rename Parameter, and Rename Variable help to remove TD while the ROs that deal with more than one file namely, Extract and Move Method, Move Class, and Move and Rename Class increases the total amount of TD. These results are aligned with previous studies (e.g., [22]), that although addressing similar research questions, focused only on code smells and performed a more coarse-grained analysis on open source projects.

This study supports, within the limits of the threats of validity, that even though ROs are thought as a means to mitigating TD, in some cases, they might contribute to the accumulation of TD if not applied with care. This does not necessarily mean that refactoring operations lower the code quality, but what our results in the analyzed context suggest is that certain operations might tend to introduce new problems in the code (i.e., TD) while might not be effective on solving the problem the developer had in mind. Further, from a managerial point of view, our findings about the impact of refactoring operations on TD can, at the same time, help reduce the waste of effort by developers.

Chapter 5

Further Investigation of the Survivability of Code Technical Debt Items

This chapter is based on the following paper:

> **Ehsan Zabardast**, Kwabena Ebo Bennin, and Javier Gonzalez-Huerta. "Further investigation of the survivability of code technical debt items." *In Journal of Software: Evolution and Process.* 34(2):e2425. Wiley (2022)[1]

[1] https://doi.org/10.1002/smr.2425

Context: Technical Debt (TD) discusses the negative impact of sub-optimal decisions to cope with the need-for-speed in software development. Code Technical Debt Items (TDI) are atomic elements of TD that can be observed in code artefacts. Empirical results on open-source systems demonstrated how code-smells, which are just one type of TDIs, are introduced and "survive" during release cycles. However, little is known about whether the results on the survivability of code-smells hold for other types of code TDIs (i.e., bugs and vulnerabilities) and in industrial settings.

Goal: Understanding the survivability of code TDIs by conducting an empirical study analysing two industrial cases and 31 open-source systems from Apache Foundation.

Method: We analysed 133,670 code TDIs (35,703 from the industrial systems) detected by SonarQube (in 193,196 commits) to assess their survivability using survivability models.

Results: In general, code TDIs tend to remain and linger for long periods in open-source systems, whereas they are removed faster in industrial systems. Code TDIs that survive over a certain threshold tend to remain much longer, which confirms previous results. Our results also suggest that bugs tend to be removed faster, while code smells and vulnerabilities tend to survive longer.

Keywords: *Survivability, Code Technical Debt Items, Code Smells, Bugs, Vulnerabilities*

5.1 Introduction

There is an ever-increasing pace in the size and complexity of software systems as they evolve, as Lehman [114, 116] formulated in his Laws of Software Evolution, also known as Lehman's Laws. As a consequence of this ever-increasing size and complexity, software systems accumulate Technical Debt as they are developed and evolve [106]. Technical Debt (TD) [55] is a metaphor commonly used to discuss the negative impact of sub-optimal design decisions, often taken to cope with the need for speed in the development. As a software system evolves, these sub-optimal design decisions taken in order to be able to deliver the product in time can potentially hinder its maintainability and even our ability to deliver future releases of the product [106].

Sub-optimal design decisions are often not visible; however, they might manifest in the form of Technical Debt Items (TDIs), which are the mani-

festations of TD [106]. TDIs can take the form of vulnerabilities and code smells [43, 107, 130], whilst some of them might be visible, and they might materialise in terms of bugs or defects [107]. TD items are "atomic elements of TD" that connect a set of artefacts (e.g., code) with the consequences of the quality [106] and have been introduced as a way to quantify or visualise TD. Bugs, code smells, and vulnerabilities are some examples of code Technical Debt Items (TDIs) [106, 128, 182]. These code TDIs go a long way to affect the development process and evolution of the software system, thus creating friction [106] and increasing the maintenance effort [192]. Adding new functionality to the existing system is also challenging when code TDIs are present in the system [154].

Several empirical studies acknowledge the negative effects of code TDIs on software quality, e.g., [48, 146]. The repaid code TD (removing code TDI) not only improves the maintainability of the software system but also reduces the maintenance costs [192]. Studying the life cycle of code TDIs (bugs, code smells, and vulnerabilities) can therefore help prioritise the maintenance activities [214]. Understanding the life cycle of code TDIs is essential in building support tools as well. The empirical analysis of the evolution and survivability (i.e., the time that the code TDIs remain in the system) of the code TDIs thus have crucial implications for software development teams.

Prior research [11, 43, 48, 60, 131, 175, 213, 214] have studied the survivability of code TDIs in software systems - mainly focusing on – some – code smells. While studies are investigating the effects of code TDIs on codebase and code entities, few empirical studies have been conducted on industrial settings. Additionally, a more in-depth investigation and insights into how development and maintenance activities impact the survivability of code TDIs are yet to be considered by prior studies. By taking inspiration and following a similar approach of the study by Tufano et al. [214], we conduct a large-scale empirical but parallel study where we focus on a different scope, a much wider set of code TDIs, and different research questions. It is important to clarify that the study by Tufano et al. [214] uses the code smells by Fowler et al. [77] whereas we study the code smells as defined and detected by SonarQube, although, of course, there is some extent of overlap among the two code smells sets.

In addition to code smells, we investigate the survivability of bugs and vulnerabilities as defined by SonarQube. Bugs are of interest since, although the bugs identified by SonarQube "can lead to errors or unexpected behaviour at

runtime."[2][3] Understanding how they are treated and how much time they survive in the system can help us understand how different types of TDI are mitigated and consider these findings to plan TD more efficient mitigation actions. Vulnerabilities are related to security issues and are usually discovered after some time has passed, making them different from code smells. Therefore, studying their survivability can give us more insight into how they are treated during the development.

This paper makes the following contributions:

- Provides insights about the survivability of code TDIs (bugs, code smells, and vulnerabilities);
- Provides a replication package for reproducible results and analysis by other researchers[4].

The rest of this article is organised as follows: Section 5.2 summarises the related works of studies. Section 5.3 describes the data collection and analysis process. The results are described in Section 5.4. Section 5.5 provides a discussion of the implications of the results. Lastly, the threats to validity and conclusions of this study are presented in Section 5.6 and Section 5.7 respectively.

5.2 Related Work

Several studies have focused on the detection and understanding of when and how the introduction of specific types of code TDI (i.e., code smells) impact software maintenance and quality (e.g., [11, 169]). These studies use historical data to evaluate the lifespan of code smells and help improve maintenance activities and code quality. Moreover, the empirical analysis of the evolution of code smells during the product life-cycle has been addressed in research studies [11, 131]. Chatzigeorgiou and Manakos [48] investigated the presence and evolution of code smells through an exploratory analysis of past versions of a software system. With a focus on when and how code smells are introduced and removed from software systems, the authors examined the evolution of three

[2] `https://docs.sonarqube.org/latest/user-guide/issues/`

[3] It is important to note that bugs, according to SonarQube terminology, are not a synonym of fault, which is a manifestation of an error in the software, but rather coding errors that *might break the code, and then manifest as a fault.*

[4] `https://github.com/ehsanzabardast/code_tdi_survivability`

types of code smells in two open-source systems. Their results indicated that most code smells last as long as the software system operates, and refactoring does not necessarily eliminate code smells. This is not unusual as a study by Peters and Zaidman [169] showed that developers, although being very aware of the presence of code smells in their code, tend to ignore the impact of those code smells on the maintainability of their code. This observation ignited interest in assessing the impact of code smells on maintenance activities within the research community, and several empirical studies have been conducted since then. Marcilio et al. [133] investigated the usage of an automatic static analysis tool (ASAT), i.e., Sonarqube, to examine its usage by the developers. They report that practitioners can benefit from using ASATs if they are properly configured, i.e., using relevant rules. Their results show that only 8.76% of the code TDIs are *fixed* from the detected code TDIs among which code smells and major issues are more prevalent.

A study by Yamashita and Moonen [223] consists of an empirical study of four Java-based systems before entering the maintenance phase and observing and interviewing 14 developers who maintained the systems. They identified 13 maintainability factors that are impacted by code smells. Further empirical studies by the same authors (Yamashita and Moonen [224]), focused on the interaction between code smells and their effect on maintenance effort, revealed that some inter-smell relations were associated with problems during maintenance and some inter-smell relations manifested across coupled artefacts. A study by Sjøberg et al. [192] demonstrated that the effect of code smells on maintenance effort was limited. Digkas et al. [61] investigate the evolution of TD in open-source systems in the Apache ecosystem over time. Their results suggest that TD increases monotonically over time in most of the investigated systems.

More studies have recently empirically analysed different types of technical debt items in several other software projects intending to understand the evolution of code smells, when and how they are introduced into systems. Tufano et al. [213] conducted an extensive empirical study on 200 open-source systems and investigated when bad smells are introduced. Comprehensive analysis of over 0.5M commits and manual analysis of 9164 smell-introducing commits revealed that smells are not introduced during evolutionary tasks. A further study by the same authors [214] contradicts common wisdom, showing that the majority of code smells are introduced when an artefact is created and not during the evolution process where several changes are made to software artefacts. They also observed that 80% of smells are not removed, and they survive as

long as the system functions, confirming previous results by Chatzigeorgiou and Manakos [48].

Additionally, some research has been conducted on how much attention is dedicated to technical debt items such as code smells, bugs, and others by software developers and how these TDIs are resolved during the evolution process of a software system. A recent work by Digkas et al. [60] analysed the life cycles of code TDIs in several open-source systems. The authors reported a case study focusing on the different types of code TDIs fixed by developers and the amount of technical debt repaid during the software evolution process. The study analysed the evolution (weekly snapshots) of 57 Java open-source software systems under the Apache ecosystem. The study revealed that allocating resources to fix a small subset of the issue types contributes towards repaying the technical debt. Similarly, the recent study of Saarimäki et al. [182] aimed to comprehend the diffuseness of TD types, how much attention was paid to TDIs by developers and how severity levels of TDIs affected their resolution. The authors observed that code smells are the most introduced TDIs, and the most severe issues are resolved faster. To understand the needs of software engineers with regards to technical debt management, Arvanitou et al. [12] surveyed 60 software engineers from 11 companies. The authors observed that developers were mostly concerned with understanding the underlying problems existing in source code, whereas managers cared most about financial concepts.

Prior studies have, in general, focused on studying the introduction and evolution of code smells in open-source software systems. To the best of our knowledge, only the work by Digkas et al. [60] and our previous work ([230]) analysed TD repayment. Digkas et al. [60] analysed TD repayment by focusing on a broader set of TDIs, but with a coarse granularity (weekly snapshots) whilst we analyse the effect at commit level. In our previous work [230], we analysed the impact of different activities on TD, i.e., whether each activity contributed to the accumulation or repayment of TD, whilst in this paper, we focus on TDIs. We present a statistical model on the survivability of code TDIs.

This study aims to extend the findings of previous studies in literature, especially the study by Tufano et al. [214]. We aim to verify and complement the results obtained in the study of Tufano et al. [214] with regards to the survivability aspects by extending the scope of the analysis considering a bigger set of code smells and other categories of code TDI that can appear in code-bases such as *bugs*, and *vulnerabilities*. We thus analyse a similar set of code TDI as the one considered by Digkas et al. [60] by focusing on individual commits.

5.3 Research Methodology

We have conducted a sample study with the aim of answering the following research question:

- **RQ.** *What are the differences in the survivability of the types of code TDI, namely bugs, code smells, and vulnerabilities?*

The purpose of a sample study is to "study the distribution of a particular characteristic in a population (of people or systems), or the correlation between two or more characteristics in a population. [196]" This sample study aims at assessing the survivability of code TDIs in large-scale industrial and open-source systems by using robust statistical tests. We conduct an empirical study to investigate further how long code TDIs in codebases survive. To achieve this goal, we have conducted a longitudinal study in two industrial and 31 open-source systems from the Apache Foundation. We analysed 133,670 code TDIs in 193,196 commits in total to investigate *the survivability of different code TDI types: bugs, code smells, and vulnerabilities.*

In the subsections below, we provide details on the data collection and analysis.

5.3.1 Context Selection

We selected two industrial systems for this study and 31 open source systems (OSS) from the Apache Software Foundation, all developed in Java. We have collected the data for this study collaborating with two large-scale companies that work in the areas of FinTech (Industrial 1 system) and banking and financial services (Industrial 2 system). The industrial systems were selected by convenience, because they are long-lived, large-scale systems that are still in production and continuously evolving and have the following characteristics:

- **Industrial 1.** The selected system has over one million lines of code and has been under development for over ten years. We analysed 9,331 commits that contained 33,974 code TDIs. The developing company of this case is a large-sized company (enterprise size), which is well versed in agile practices. The development of the selected system followed craftmanship practices during the studied period.

- **Industrial 2.** The selected system has over 60,000 lines of code and has been under development for over four years. We analysed 8,414 commits

that contained 1,729 code TDIs. The developers are using SonarQube to keep track of the introduction of TDIs and the growth of TD in this system. The developing company of this case is a medium-sized company (enterprise size), which is well versed in agile practices.

The companies and systems were selected by convenience and availability. The partner companies are mature in their development practices and have well-established, successful products. They are interested in continuously improving their products and development life-cycles, thus their willingness on participating in studies like this. All the collaborating companies work on developing software-intensive products and services.

The Industrial systems have been analysed commit by commit. For each commit, we examine the code TDI and trace them throughout the analysed period with their key IDs. This ensures that there are no repeated code TDI in the collected data. The industrial system 1 is a huge system with thousands of commits. There are many new issues that are created throughout the studied commits. This provided us with many code TDI instances.

Table 5.1 summarises the historical information of the OSS analysed systems. The industrial systems and all the OSS systems were hosted in `git` repositories[5] and the SonarQube data was collected through a web API available online[6]. The data collection for the industrial cases was performed using the corresponding SonarQube API in their on-site SonarQube installations.

5.3.2 Using SonarQube for Code TDI Detection

While there are alternative tools to detect the TDIs in the codebase of a system such as Codacy[7] and PMD source code analyser[8], we decided to use SonarQube[9] because it is widely used in both industrial and open-source systems [182] and has been used in other research studies, e.g., Zabardast et al. [230], Digkas et al. [60], and Guaman et al. [87]. SonarQube, similar to other static analysis tools, parses the code base, and builds a model for each commit being analysed.

[5] `https://github.com/apache`
[6] `https://sonarcloud.io/organizations/apache/projects`
[7] `https://www.codacy.com/`
[8] `https://pmd.github.io/pmd-6.17.0/index.html`
[9] `https://www.sonarqube.org/`

Table 5.1: General description of the analysed open-source systems.

	System Name	System Size	Number of			
			Commits	TDIs	Classes	Contributors
1	Ant (v 1.10.0)	12503	14698	9351	1321	53
2	Commons Compress (v 1.21)	26974	3133	772	377	56
3	Commons Geometry (v 1.1)	16483	431	99	355	10
4	CXF (v 3.5.0)	427463	16155	10000§	7555	154
5	Groov (v 3.0.9)	189114	18181	10000§	1724	307
6	Hadoop Ozone (v 1.2.0)	132255	3202	3531	2182	104
7	IoTDB Project (v 0.13.0)	120051	4611	2082	1685	94
8	Isis (Aggregator) (v 2.0.0)	165713	15676	5532	5232	40
9	Jackrabbit FileVault (v 3.5.1)	47692	8859	2053	3128	23
10	JSPWiki (v 2.11.0)	56647	8853	3598	618	13
11	Karaf (v 4.3.1)	123212	8482	6567	1582	141
12	PDFBox (v 3.0.0)	141971	9694	1876	1348	6
13	POI (version unspecified)	259765	10679	10000§	3512	15
14	Ratis (v 2.2.0)	38622	1131	957	607	41
15	ServiceComb Pack (v 0.7.0)	17494	1568	614	458	55
16	ServiceComb Toolkit (v 0.3.0)	15262	237	331	639	5
17	Shiro (v 2.0.0)	32723	2125	1964	724	46
18	Sling - CMS (v 1.0.5)	11409	894	52	287	9
19	Sling Distribution Core (v 0.4.3)	14185	483	621	267	9
20	Sling Launchpad Integration Tests (v 11)	16180	483	1901	167	17
21	Sling Resource Resolver (v 1.7.11)	10377	710	444	101	24
22	Sling Scripting JSP (v 2.5.5)	27233	275	1929	124	12
23	Incubator Tamaya (Retired) (v 0.4)	19056	1618	702	460	10
24	Dolphin Scheduler (v 1.3.6)	58979	4441	2135	912	175
25	Gateway (v 1.6.0)	76495	2455	2241	1479	54
26	Hop Orchestration Platform (v 1.0)	483430	1572	10000§	3240	20
27	Jmeter (v 5.5)	117352	17331	4362	1386	35
28	Openmeetings (v 7.0.0)	99052	3121	1439	584	12
29	PLC4X (v 0.9.0)	57162	3740	2491	1007	43
30	Roller (v 6.1.0)	66248	4549	3318	610	12
31	Struts 2 (v 2.6)	110941	6064	8142	2060	51
	Totals	2992043	175451	109104	45731	-

§ The limit for fetching the data from API is 10000 code TDIs.

SonarQube classifies code TDIs into three types of *Bugs*, *Code Smells*, and *Vulnerabilities*. SonarQube's definitions[10] for these types are presented below with one example for each type[11].

- **Bug**: SonarQube defines bugs as "a coding error that will break your code and needs to be fixed immediately". It is important to clarify that SonarQube *bugs* are not what it is reported in issue tracking systems, but rather detected through its static analysis. SonarQube's definition contrasts with the most spread definition of **Bug**: a synonym of fault, which is a manifestation of an error in the software, such an incorrect step, process, or data definition in a computer program [91].

- **Code Smell**: SonarQube defines Code Smells as a maintainability issue that makes your code confusing and difficult to maintain. **Code Smells** are commonly defined as surface indications of deeper problems in the system [76, 77], and they are "sniffable" at code level [74]. Code-Smells are sometimes also referred to as Code Anti-Patterns, which are common re-occurring solutions to a problem, which generate negative consequences [44].

- **Vulnerability**: SonarQube defines Vulnerabilities as a point in your code that's open to attack. **Vulnerabilities** are usually referred as a weakness in an information system, system security procedures, internal controls, or implementation that could be exploited by a threat source [157]. A vulnerability "does not cause harm in itself as there needs to be a threat present to exploit it [90]".

We want to emphasise that SonarQube's definitions of the code TDI types should be considered in its own context. In this paper, when we refer to the above-mentioned TDI types, we refer to SonarQube definitions.

SonarQube is a static analysis tool that detects the aforementioned code TDI categories using rules. Although bugs, vulnerabilities, and code smells in SonarQube are defined on static properties of the code, their detection rely on the existence of detection rules. It is possible to add new rules, and new rules are also included when the tool is upgraded. Therefore, a code TDI can be detected on a given location of the code when that portion of code has been created or modified (at commit time) if the rule exists. However, it is also possible that we detect a code TDI in retrospective, i.e., new rules trigger the detection of a

[10] `https://docs.sonarqube.org/latest/user-guide/issues/`
[11] All the SonarQube rules can be browsed at `https://rules.sonarsource.com`

code TDI in a portion of code that was created before that particular rule was created. Therefore, the developer of that particular modification of the code base might not have been aware of the introduction of the code TDI at commit time.

The rest of this subsection is dedicated to providing examples for each type of code TDI from SonarQube.

Bug - Regex lookahead assertions should not be contradictory

Lookahead assertions are a regex feature that makes it possible to look ahead in the input without consuming it. It is often used at the end of regular expressions to make sure that substrings only match when they are followed by a specific pattern.

However, they can also be used in the middle (or at the beginning) of a regex. In that case there is the possibility that what comes after the lookahead does not match the pattern inside the lookahead. This makes the lookahead impossible to match and is a sign that there's a mistake in the regular expression that should be fixed.

Noncompliant Code Example:

```
Pattern.compile("(?=a)b"); // Noncompliant, the same
    character can't be equal to 'a' and 'b' at the same
    time
```

Compliant Solution:

```
Pattern.compile("(?<=a)b");
Pattern.compile("a(?=b)");
```

Code Smell - Methods returns should not be invariant

When a method is designed to return an invariant value, it may be poor design, but it should not adversely affect the outcome of your program. However, when it happens on all paths through the logic, it is surely a bug.

This rule raises an issue when a method contains several return statements that all return the same value.

Noncompliant Code Example:

```
int foo(int a) {
  int b = 12;
  if (a == 1) {
    return b;
  }
  return b; // Noncompliant
}
```

Vulnerability - Server certificates should be verified during SSL/TLS connections

Validation of X.509 certificates is essential to create secure SSL/TLS sessions not vulnerable to man-in-the-middle attacks.

The certificate chain validation includes these steps:

- The certificate is issued by its parent Certificate Authority or the root CA trusted by the system.
- Each CA is allowed to issue certificates.
- Each certificate in the chain is not expired.

This rule raises an issue when an implementation of X509TrustManager is not controlling the validity of the certificate (ie: no exception is raised). Empty implementations of the `X509TrustManager` interface are often created to disable certificate validation. The correct solution is to provide an appropriate trust store.

```
class TrustAllManager implements X509TrustManager {

    @Override
    public void checkClientTrusted(X509Certificate[]
        chain, String authType) throws
        CertificateException { // Noncompliant, nothing
        means trust any client
    }
```

```
    @Override
    public void checkServerTrusted(X509Certificate[]
        chain, String authType) throws
        CertificateException { // Noncompliant, this
        method never throws exception, it means trust
        any server
        LOG.log(Level.SEVERE, ERROR_MESSAGE);
    }

    @Override
    public X509Certificate[] getAcceptedIssuers() {
        return null;
    }
}
```

5.3.3 Data Analysis

To understand the survivability of each code TDI's in the studied systems, we analyse the existence of code TDIs from their introduction in the codebase until they are marked as "closed". Similarly, the code TDIs that are still remaining in the system are marked as "open". In our analysis, we include *all* the code TDIs present in the system, i.e., we include the code TDIs marked as closed and open. Figure 5.1 summarises the data analysis procedure for this study. We use the collected data from SonarQube API (Step 1). We process the preprocess of the data to extract the number of survived days and commits for each code TDI (Step 2). The processed data is used for the survival analysis (Step 3). The details of the analysis are provided in the rest of this subsection.

We use the creation time and the updated time – when the TDI was closed – of each TDI in the collected data. The extracted information is used to calculate the number of survived days and the number of survived commits for each case.

We use *The Number of Survived Days* and *The Number of Survived Commits* similar to the study designed by Tufano et al. [214] as complementary metrics to capture the views of code TDI survivability. Considering these metrics individually might be misleading since a project can be inactive for months (i.e., nothing committed for a while). The processed data will contain code TDI with no removal time, i.e., the code TDIs that are not marked as "closed" in the

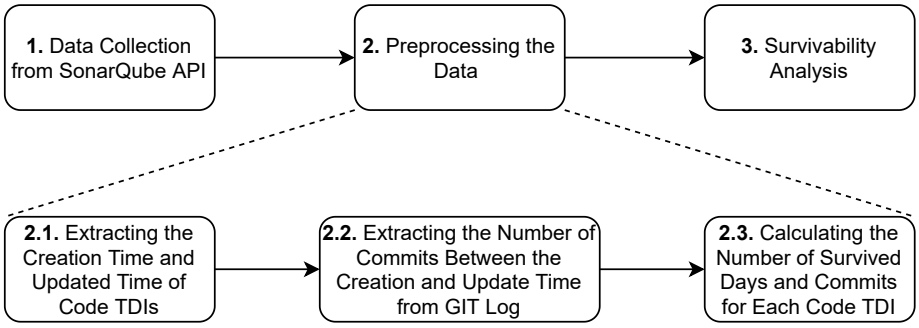

Figure 5.1: Data analysis process.

collected data. To interpret this, we use the same approach as [214] and mark such code TDI as "*Censored Data*".

We analyse the data using Survival analysis. Survival analysis is a statistical method that analyses and models the duration of events until other events happen [149], i.e., code TDI removal in this case. The survival function for code TDI, $S(t) = Pr(TDI > t)$ indicates that a code TDI exists longer than a specific time t. The survival analysis creates a Survivability Model based on historical data. Using this model, we can generate survival curves illustrating the survival probability as a function of time. The model can handle both *complete data* (observations with an ending event) and *incomplete data* (observations without an ending event) if the data is marked properly. We create survival models based on the number of survived days and the number of survived commits for each type of code TDI, namely bugs, code smells, and vulnerabilities. The analysis is done in `R` using the `survival`[12] and `survminer`[13] packages. The `Surv` function is used to generate the survival model, and the `survfit` function is used to estimate survival curves. Additionally, similar to [214], we utilise the Kaplan-Meier estimator [96] in the analysis to estimate the removal time for the *incomplete* observations, therefore we also included "censored" data-points.

[12] `https://cran.r-project.org/web/packages/survival/index.html`
[13] `https://cran.r-project.org/web/packages/survminer/index.html`

5.4 Results

In this section, we report the results of our study. We focus on the three types of code TDI (i.e., code smells, bugs, and vulnerabilities), as done in [60]. The raw experimental results, data sets extracted from open-source systems and the source code for implementing the experiments is provided in our replication kit online[14].

To address the research question, we analysed the survivability of the code TDIs collected from two industrial systems and 31 open-source systems from the Apache Foundation. We have analysed the survivability both in terms of the survived days and survived commits. Table 5.2 summarises the total code TDI identified per system in the industrial systems and the aggregated numbers of identified code TDI for open-source systems. Figure 5.2 illustrates the box plots for the distribution of the survived days (left) and commits (right) for the detected code TDIs. The first and second rows belong to the industrial systems, and the third row belongs to the open-source systems. The plots are presented on a log scale to make the representation easier to read. We observe that the medians of the distributions are not significantly different across different types of code TDI for the number of survived days and the number of survived commits. However, they are much lower in the industrial systems for all types of code TDIs.

Table 5.2: The total code TDI identified per system in the industrial systems and the aggregated numbers of identified code TDI for open-source systems.

TDI Type	**Industrial 1**	**Industrial 2**	**Open-Source (Aggregated for 31 OSS)**	**Total**
Bugs	31,808	90	3,625	35,523
Code Smells	1,266	1,639	93,012	95,917
Vulnerabilities	900	0	1330	2,230
Total	33,974	1,729	97,967	133,670

In the case of Industrial 2, SonarQube did not detect any vulnerabilities (marked as "No Observations" in Figure 5.2). For this particular system, we

[14] `https://github.com/ehsanzabardast/code_tdi_survivability`

have analysed the last two years only. The fact that vulnerabilities are often reported only until some time has passed might explain why we have not found any vulnerabilities in this particular case. Similarly, in Tables 5.3 and 5.3, the presented descriptive statistics, for the case of industrial 2, are only for code smells and bugs, and they should be interpreted keeping this in mind.

Tables 5.3 and 5.3 summarise the descriptive statistics for the number of survived days and commits for all systems. The table distinguishes between the industrial systems and open-source systems for each row labelled as *Industry* and *Open-Source*. We use the median as the measure of central tendency to minimise the effect of outliers [126]. We use median values as the references of analysis, as summarised in Tables 5.3 and 5.3. Note that the data presented in Figure 5.2 and Tables 5.3 and 5.3 are only representative of the cases with terminal event occurred.

We observe that the majority of the code TDIs detected in the industrial 1 are removed before the 43^{rd} day (before 58^{th} commit), and the majority of the code TDIs detected in the industrial 2 are removed before the 580^{th} day (before 496^{th} commit). In open-source systems, the majority of the code TDIs are removed before the 398^{th} day (before 722^{nd} commit).

Figure 5.3 illustrates the calculated survival probability curves for all the systems (survived days on the left and survived commits on the right). The first row belongs to the industrial 1 system, the second row belongs to the industrial 2 system, and the third row belongs to the open-source systems.

Our first observation is that the survivability of code TDIs vary in the systems under investigation, both in terms of the number of days and the number of commits. Considering these calculated probability curves, the survival probability of code TDIs in terms of survived days is highest in open-source systems. In contrast, the survival probability of code TDIs in industrial systems is lower. Having a higher survivability probability in terms of days means that for the same number of days, code TDIs in open-source systems have a higher probability of surviving as compared to the probability of a code TDI surviving with the same number of days in the industrial systems. In other words, we have observed that in the analysed systems, code TDIs tend to be removed faster, in terms of the number of days, in industrial settings.

We also observe that the survival probability of code TDIs in terms of survived commits is higher in open-source systems. Having a higher survivability

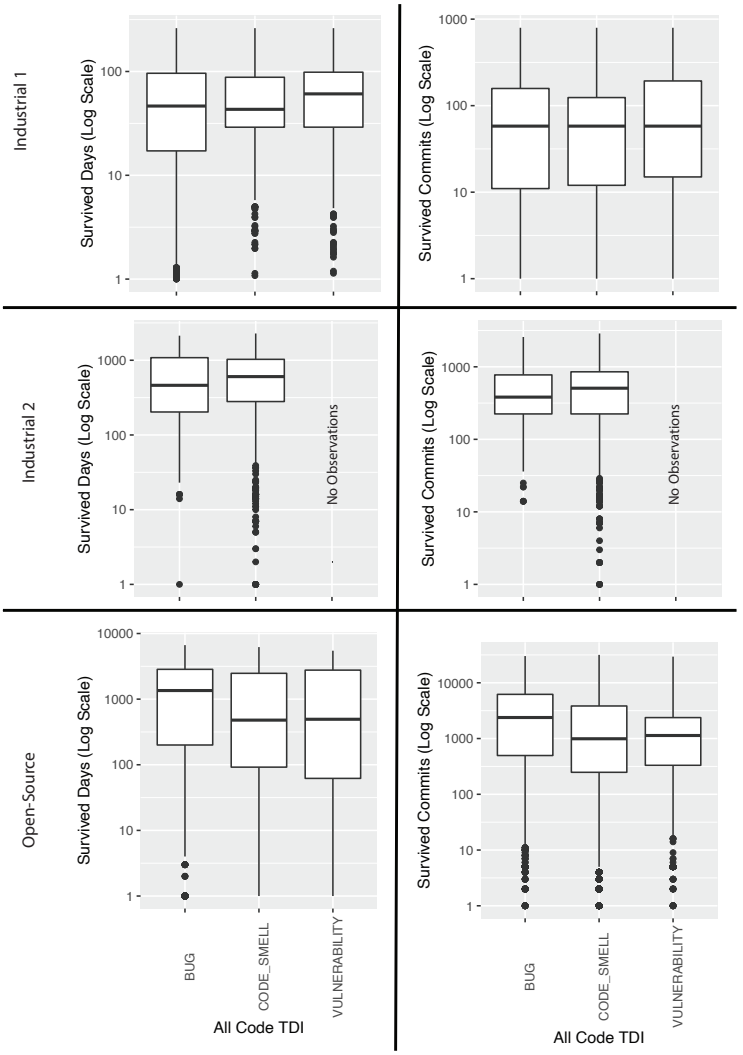

Figure 5.2: Distribution of the number of days (left) and commits (right) that code TDIs survived. The first row belongs to the industrial system and the second row belongs to the open-source systems. Note that the data presented in this figure is representative only for the cases with terminal event occurred.

Table 5.3: Descriptive statistics for the number of survived days and commits that code TDIs survived in the systems. Note that the data presented in this figure is only representative of the cases with terminal event occurred. [The original table is divided into two tables because of page limitation – Tables 4.3 and Table 4.4]

Case	Min	1stQu.	Median	Mean	3rdQu.	Max
			Survived Days			
Industry 1 - All Code TDI	0	12	43	58.70	95	262
Industry 2 - All Code TDI	0	221	580	678.01	1,007	2,283
Open-Source - All Code TDI	0	21	398	1,255.80	2,370	6,656
Industry 1 - Bug	0	11	43	58.37	96	262
Industry 2 - Bug	1	203	462	665.81	1,083	2,134
Open-Source - Bug	0	168	1,287	1,662.78	2,767	6,656
Industry 1 - Code Smell	0	15	43	62.32	83	252
Industry 2 - Code Smell	0	221	584	679.16	989	2,283
Open-Source - Code Smell	0	18	398	1,238.73	2,338	6,211
Industry 1 - Vulnerability	0	23	49	1,340.52	98	261
Industry 2 - Vulnerability	-	-	-	-	-	-
Open-Source - Vulnerability	0	26	415	1,340.52	2,767	5,470

Table 5.4: Descriptive statistics for the number of survived days and commits that code TDIs survived in the systems. Note that the data presented in this figure is only representative of the cases with terminal event occurred. [The original table is divided into two tables because of page limitation – Tables 5.3 and Table 5.4]

Case	Min	1stQu.	Median	Mean	3rdQu.	Max
			Survived Days			
			Survived Commits			
Industry 1 - All Code TDI	1	11	58	131.72	155	795
Industry 2 - All Code TDI	1	200	496	603.29	813	2,881
Open-Source - All Code TDI	1	68	722	3,154.40	3,492	31,690
Industry 1 - Bug	1	11	58	131.11	158	795
Industry 2 - Bug	1	224	375	545.30	773	2577
Open-Source - Bug	1	255	1,719	4,443.13	4,950	30,185
Industry 1 - Code Smell	1	12	58	143.38	124	695
Industry 2 - Code Smell	1	198	498	606.80	813	2,881
Open-Source - Code Smell	1	67	707	3,114.26	3,311	31,690
Industry 1 - Vulnerability	1	15	58	138.74	193	790
Industry 2 - Vulnerability	-	-	-	-	-	-
Open-Source - Vulnerability	1	186	1,134	2,448.88	2,376	29,493

probability in terms of commits means that for the same number of commits, code TDIs in open-source systems have a higher probability of surviving as compared to the probability of a code TDI surviving with the same number of commits in the industrial systems. In other words, we have observed that in the analysed systems, code TDIs tend to be removed faster, in terms of the number of commits, in industrial settings.

We use 100 days as the point of reference to present the results of survivability models. However, instead of using 10 commits as in Tufano et al. [214] work, we use 100 commits in our analysis as the point of reference. This is owing to the fact that the 10 commit threshold might be too short for our data sets, and the results might turn inconclusive since the probabilities of survivability for ten days are very high and very similar among systems.

We observe that, in general, code TDIs are removed faster in the investigated industrial system as compared to the investigated open-source systems. Table 5.5 summarises the probability of code TDIs surviving up to 100 days and commits for *Bug*, *Code Smell*, and *Vulnerability* separately for the industrial system and open-source systems. Note that there was no observation as *vulnerability* in the Industrial 2. Therefore we cannot be presenting any survival probability in Table 5.5 for Industrial 2.

From the survivability models for all code TDIs in the investigated systems in Table 5.5, we observe that:

- **In the industrial 1 system:** There is a higher chance for code TDIs to be removed before 100 days (100 commits), i.e., many of the code TDIs are removed relatively soon.
  - *Bug:* There is a 18.476% chance that the investigated code TDIs survive until 100 days (38.312% chance that the investigated code TDIs survive until 100 commits).
  - *Code Smell:* There is a 13.850% chance that the investigated code TDIs survive until 100 days (37.180% chance that the investigated code TDIs survive until 100 commits).
  - *Vulnerability:* There is a 23.200% chance that the investigated code TDIs survive until 100 days (39.890% chance that the investigated code TDIs survive until 100 commits).
- **In the industrial 2 system:** There is a lower chance for code TDIs to be removed before 100 days (100 commits).

Table 5.5: The probability of code TDIs surviving up to 100 days and commits.

Case	Survival Probability	Standard Error	Lower 95% CI	Upper 95% CI
		Survived Days		
Industry 1 - Bug	18.48%	0.00265	0.17963	0.19002
Industry 2 - Bug	100%	0	1	1
Open-Source - Bug	91.84%	0.00474	0.90913	0.92773
Industry 1 - Code Smell	13.85%	0.01240	0.11620	0.16500
Industry 2 - Code Smell	99.41%	0.00221	0.98982	0.998448
Open-Source - Code Smell	94.47%	0.00084	0.94306	0.94635
Industry 1 - Vulnerability	23.20%	0.01770	0.19980	0.26940
Industry 2 - Vulnerability	-	-	-	-
Open-Source - Vulnerability	99.38%	0.00236	0.98920	0.99844
		Survived Commits		
Industry 1 - Bug	38.31%	0.00332	0.37667	0.38968
Industry 1 - Bug	98.36%	0.01630	0.95230	1.0000
Open-Source - Bug	92.61%	0.00447	0.91747	0.93499
Industry 1 - Code Smell	37.18%	0.01730	0.33940	0.40730
Industry 2 - Code Smell	99.32%	0.00238	0.98862	0.99793
Open-Source - Code Smell	95.23%	0.00078	0.95074	0.95380
Industry 1 - Vulnerability	39.89%	0.02050	0.36070	0.44130
Industry 2 - Vulnerability	-	-	-	-
Open-Source - Vulnerability	99.68%	0.00158	0.99374	0.99994

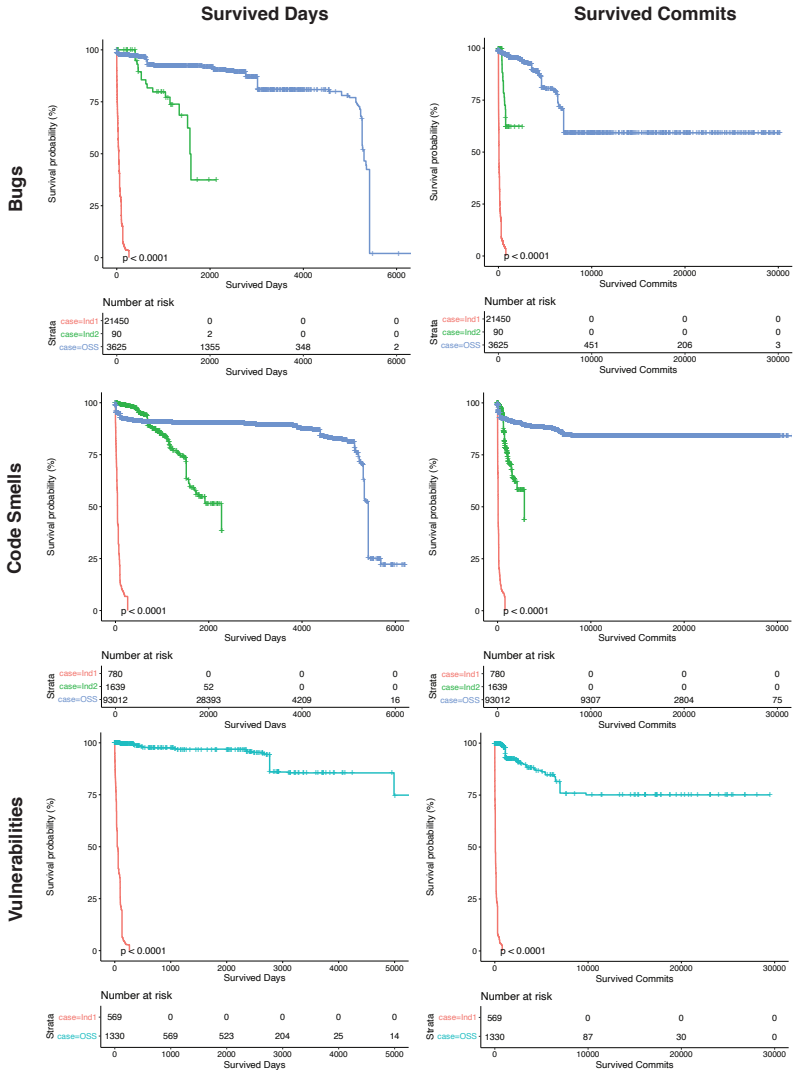

Figure 5.3: Distribution of the number of days (left) and commits (right) that code TDIs survived in all systems.

  - *Bug:* There is a 100% chance that the investigated code TDIs survive until 100 days (98.360% chance that the investigated code TDIs survive until 100 commits).
  - *Code Smell:* There is a 99.410% chance that the investigated code TDIs survive until 100 days (99.320% chance that the investigated code TDIs survive until 100 commits).
  - *Vulnerability:* There are not enough observations to calculate survival probabilities.

- **In open-source systems:** Similar to the industrial systems, there is a higher chance for code TDIs to be removed before 100 days (100 commits), i.e., many of the code TDIs are removed relatively soon but if they are not removed, they can stay in the system for as long as the system is in production.
  - *Bug:* There is a 91.838% chance that the investigated code TDIs survive until 100 days (92.619% chance that the investigated code TDIs survive until 100 commits).
  - *Code Smell:* There is a 94.470% chance that the investigated code TDIs survive until 100 days (95.227% chance that the investigated code TDIs survive until 100 commits).
  - *Vulnerability:* There is a 99.381% chance that the investigated code TDIs survive until 100 days (99.683% chance that the investigated code TDIs survive until 100 commits).

We analysed the different types of code TDIs individually for the industrial system and open-source systems. Individual box plots for the distribution of the survival days (left) and commits (right) for the different types of code TDI detected in the industrial systems, and open-source systems are presented in Figure 5.2. We observe that the median of the distribution for bug, code smell, and vulnerability is significantly higher in the open-source systems when compared to the industrial systems both for the number of survived days and the number of survived commits.

Finally, in order to shed light on the nature of code TDI removal, we have analysed a selection of commits in Industry 1. This system was the subject of our analysis in our previous work [230], since we can extract the main intention behind the commits. We extracted this information by analysing the tags on the commits. The developers systematically tagged the commits with the main purpose of that commit. Out of the analysed commits for Industry 1 system,

we extracted the main intention of 2,286 commits. We looked for three actions namely *new development* with 801 commits, *refactoring* with 476 commits, and *bug fixing* with 226 commits. 783 commits were tagged with other intentions such as build and merge. There are 2,506 code TDIs removed in the investigated commits. Commits tagged as *new development* are responsible for the removal of 596 code TDI, Commits tagged as *refactoring* are responsible for the removal of 649 code TDIs, and commits tagged as *bug fixing* are responsible for the removal of 39 cases. Therefore, out of 2,506 cases of code TDI removal, 688 (27.45%) cases were removed because of *refactoring* and *bug fixing*. We also observe that 596 (23.78%) cases were removed because of *new development*. Previous research (e.g., [99, 230]) found that major refactorings can be embedded in the development of new features, and therefore can be the cause for the removal of TDIs.

5.5 Discussion

5.5.1 General Findings

Our empirical analyses reveal the unpredictable nature of the survivability of code TDIs in software systems. The survivability of the issues ranges from 0 to 2,283 days for the industrial systems and 0 to 6,656 days for open-source systems.

The median for the industrial system stay below 100 days; therefore, the TD mitigation strategies to remove code TDIs seem to be more effective in industrial systems as compared to open-source systems. Our results also suggest that, for open-source systems, when code TDIs remain in the system after 100 days, they can survive for a long time in the system. Similar behaviour can be observed when analysing the number of survived commits, which ranges from 1 to 2,881 commits for the industrial systems and 1 to 31,690 commits for open-source systems. Our results are aligned with the findings of Digkas et al. [61]. In their paper, the authors present the monotonic upward trend growing TD over time in open source systems from the Apache Software Foundation. The monotonic upward trend is a sign that the code TDIs survive for longer periods in the systems under investigation.

TD mitigation strategies seem to be more effective in removing TDIs in industrial settings in terms of the number of survived commits as well. As discussed before, regardless of the system under study, code TDIs that survive

past the 100^{th} day threshold might survive much longer, confirming the finding in the study by Tufano et al. [214].

However, we observed patterns that contrast what was found in previous research studies. We found that:

- After 100 days:
  - The survival probability for the industrial 1 system is as follows: *Bug* 17.48% - *Code Smell* 13.85% - *Vulnerability* 23.20%.
  - The survival probability for the industrial 2 system is as follows: *Bug* 100.00% - *Code Smell* 99.41%.
  - The survival probability for the open-source systems is as follows: *Bug* 91.84% - *Code Smell* 94.47% - *Vulnerability* 99.38%.
- After 100 commits:
  - The survival probability for the industrial 1 system is as follows: *Bug* 38.31% - *Code Smell* 37.18% - *Vulnerability* 39.89%.
  - The survival probability for the industrial 2 system is as follows: *Bug* 98.36% - *Code Smell* 99.32%.
  - The survival probability for the open-source systems is as follows: *Bug* 92.62% - *Code Smell* 95.23% - *Vulnerability* 99.68%.

The study by Tufano et al. [214] found that it was after 1000 days when the survival probability achieved similar values. This might be owing to the fact that we have extended the scope of our analysis to include a much wider set of code smells (the study by Tufano et al. [214] only studied five code-smells), and we also analysed bugs and vulnerabilities, which were not in the scope of previous research studies. We hypothesise that some of the five code smells analysed in the previous study by Tufano et al. [214] might not be the priority for developers in their maintenance activities, in line with what is found in [164]. Furthermore, we have observed a completely different behaviour when it comes to code-smells. Chatzigeorgiou and Manakos [48] observed that code smells are never removed and stay as long as the software system operates. Similarly, Marcilio et al. [133] observed that a low percentage (8.76%) of the code TDIs, including bugs, code smells, and vulnerabilities, are removed, suggesting that not all code TDIs detected by SonarQube are relevant to the developers.

In the case of the industrial 1 system, we are aware that the development team put an emphasis on clean code practices. This might explain the faster rate

of removal of code TDIs as illustrated in Figure 5.3. The results of our work were shared and discussed with the developer of the industrial 2 system. Throughout our discussions with the developers of the industry 2 system, they have informed us that their approach is to using SonarQube during the development. They do not use SonarQube to fix the existing problems, but they prevent new Code TDIs from arriving at the system.

5.5.2 Implications of the Results

Technical Debt (TD) management has recently been the focus of attention in academic and industrial communities [178]. The research on TD is in its initial phase, with researchers focusing on a few types of debt [4, 128]. Empirical evaluation of TD management activities, especially the evidence from the software industry, is essential to shedding light on technical debt prioritisation activities [178]. The results and the analysis methods provided by our study can help the research and industrial communities to better understand what the lifespan of different types of code TDIs is, not only code smells, and invite researchers to further investigate TD prioritisation and the activities related to it. Moreover, analysing the survivability of code TDIs in a system can be performed on fine-grained code TDIs and not just the main types, e.g., specific code-smells in isolation. The survival analysis can help the developers in two ways. First, by helping them become aware of the survivability of existing code TDIs to repay them eventually. Second, by helping them prioritise action plans to prevent the accumulation of similar code TDIs similar to the case of Industrial 2 system's developers.

SonarQube use and its impact on the survivability of the code TDI

The use of automatic static analysis tools has become popular in the last few years [133]. Given the fact that developers working on the Industrial 2 system have been using SonarQube during the last year of the development, we have examined if the use of such tools impacts the survivability of code TDIs by manually investigating the code TDIs that were created in the last year. It seems that the survivability of the code TDIs, both in terms of the number of survived days and commits, are not to be impacted by using SonarQube and follow a similar pattern to what is portraited in Figure 5.3 for the Industrial 2 system. The code TDIs that were introduced in the last year during the period where developers used SonarQube have a similar number of survival days and commits.

System type impacts the survivability of the code TDIs.

Our study suggests that the survival probability of the code TDIs, both in terms of the number of survived days and commits, varies throughout the software systems under investigation. By comparing the density distribution for all code TDIs of survived days with the density distribution for all code TDIs of survived commits presented in Figure 5.2, we observe that the industrial and open-source systems have different distributions, but these distributions follow a similar trend. The code TDIs have a similar distribution in terms of the number of days but different survival probabilities in terms of the number of commits. This might be owing to the fact that the open-source systems have more frequent commits as compared to the industrial systems. Therefore, the code TDIs are addressed in later commits, whereas in industrial systems, the code TDIs tend to be addressed and resolved closer to the point when they are introduced in the system.

Code TDIs in the industrial systems are removed faster in terms of the number of commits.

Our analysis reveals that code TDIs survive longer in open-source systems as compared to industrial projects. A viable reason can be that the quality standards in the industrial systems prevent code TDIs from staying in the system for longer periods. There are other factors that might affect how long the code TDIs survive in different systems, such as the development practices put in place, the domain, the business model, the product maturity, and the expertise of the development team, and as discussed above, the usage of static analysis tools like SonarQube. The circumstances of each system might affect the survivability of the code TDIs, e.g., the industrial systems might have more rigorous development processes, including more stringent code reviews and test processes before code is pushed into production.

Before drawing firm conclusions, each system should be analysed in isolation, considering additional factors. These factors might include the development process, developers' experience (in general and in the system), team's culture, product maturity, specific refactoring policies, developers' perception of whether the code contains code TDIs or not, and the willingness of developers to fix code TDIs.

5.6 Threats To Validity

In this section, we present the potential threats to validity that might affect the results and findings of this study. We discuss below the threats to the construct, internal, and external validity of the study.

The main threat to validity is the *Construct Validity*, i.e., the relationship between the theory and observation. The construct validity threats comprise of the errors and imprecision in measurement procedure adopted during the data collection process and whether the measurements actually reflect the construct being studied. We use SonarQube, a widely used tool for measuring TD, to detect code TDIs in the system. We also use the categories defined by SonarQube and employed in other research studies (e.g., [60]), to categorise the types of code TDIs. We identify code TDIs in the systems using the default profile for `Java` by SonarQube. We acknowledge the problems that might arise due to the use of a particular tool, i.e., SonarQube, which includes the thresholds, measurements, and rules used to detect code TDIs and the possibility of having false positives and false negatives in the collected data.

Another threat to validity is regarding our process to detect code TDIs. Code TDIs detected by our analysis in a given commit might not have been detected when the target code was developed. New rules might have been introduced to operationalise a code TDI that was unknown when the code was originally committed (this might be the case for bugs and vulnerabilities). Therefore, there would be no way for the developers to be aware of their existence when they originally committed the code. However, no tool could warn the developers to remove flaws that were not known when they were introduced. We also need to point out that the *bugs* that we focus on in this study are not faults, but they can lead to errors and unexpected behaviour at runtime.

Internal Validity refers to the degree to which the presented evidence can support a cause and effect relationship within the context of the study.

Our results are based on the `R` packages used to calculate the survivability curves. Different implementations of the same survivability analysis might lead to obtaining different results. The fact that one of the systems (i.e., Industrial 1) was developed following the principles of clean-code [134] might have an influence on the results of the analysis of how code TDIs are removed, especially when it comes to the analysis of code smells and bugs being removed due to the development of new features.

Another threat to validity is regarding the detection of closed code TDIs. There is no way for us to detect the code TDIs that are marked as “closed”

purposefully. There might be cases where code TDIs are marked as closed because a method or class is removed from the source code.

External Validity refers to the degree to which the results can be generalised. It is important to highlight that the nature of our study does not allow for statistical generalisability.

In this study, we analysed 33 systems, which are mainly Java-based software systems, two industrial systems, and thirty-one open-source systems from the Apache Software Foundation. We limit the results of this study to the systems under investigation. We understand and acknowledge that the generalisability of the results is limited and the results are representative of the *universe*, *dimensions*, and *configuration* covered in our study [152]. We can only claim that the results are applicable to the analysed commits in the systems under investigation. Moreover, we also understand that the low number of *vulnerabilities* detected in *Industry 2* and, in general, *bugs* in industrial systems make the results regarding these two types of TDIs limited. Anyway, the nature of the study with regards to industrial systems (i.e., with only two cases) suffers from the same type of constraints as case study research. In the industrial cases, where there are low numbers of bugs and vulnerabilities, we have focused on analytical generalisability [70] instead of trying to achieve statistical generalisability. Even for the OSS cases, it is not possible to aim for statistical generalisability. We present the results for the studied cases and discuss the peculiarities observed in the different systems for the different code TDIs.

We acknowledge that the removal of the code TDIs and the categories are highly dependent on each system, i.e., the development process, the developers' experience both in the software system project and their overall experience, the team's culture, the stage of development, and other factors that affect the survivability of code TDIs given a system [214]. Our analysis and findings are thus impacted by the system type under consideration.

5.7 Conclusions

In this paper, we present the results of a sample study on the survivability of code TDIs in software systems. This paper presents a study on the change history of five software systems, and it aims to understand the survivability of code technical debt items (TDIs) in the codebase. Furthermore, this study aims to extend the results of prior studies by including smells and other types of code TDIs, such as bugs and vulnerabilities in the analysis code. Therefore, we have

focused on examining and assessing the differences in survivability among the three categories of detected code TDI.

We have conducted a comprehensive empirical sample study using data from other software systems, including two large industrial software systems and 31 open-source systems from the Apache Foundation. As illustrated by the survival curves, most of the code TDI in the investigated systems are removed rather quickly, i.e., there is a 23.20% chance that they survive until 100 days. Any code TDI that survives this threshold has a higher probability of surviving longer in the system.

When the system type and commit activities are taken into account, the code TDIs in the systems which have a bigger size tend to have a longer survival duration. On the other hand, the code TDIs that survive past the median threshold tend to stay in the system for a long time.

Our findings open the door for further studies. Our results can be strengthened by digging into the other factors that affect the systems. Additionally, we believe replications are needed to strengthen the results and study whether the results might be generalisable to similar systems.

Chapter 6

Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution

This chapter is based on the following paper:

> **Ehsan Zabardast**, Javier Gonzalez-Huerta, and Binish Tanveer. "Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution" *In 2022 19th International Conference on Software Architecture Companion (ICSA-C)* (pp. 30-34). IEEE (2022).[1][2][3]

[1] *© 2022 IEEE. Reprinted, with permission, from Ehsan Zabardast, Javier Gonzalez-Huerta, and Binish Tanveer "Ownership vs Contribution: Investigating the Alignment Between Ownership and Contribution", 19th International Conference on Software Architecture Companion (ICSA-C), 2022*

[2] https://doi.org/10.1109/ICSA-C54293.2022.00013

[3] *In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of BTHs products or services. Internal or personal use of this material is permitted.*

Software development is a collaborative endeavour. Organisations that develop software assign modules to different teams, i.e., teams own their modules and are responsible for them. These modules are rarely isolated, meaning that there exist dependencies among them. Therefore, other teams might often contribute to developing modules they do not own. The contribution can be, among other types, in the form of code authorship, code review, and issue detection. This research presents a model to investigate the alignment between module ownership and contribution and the preliminary results of an industrial case study to evaluate the model in practice. Our model uses seven metrics to assess teams' contributions. Initial results suggest that the model correctly identifies misalignment between ownership and contribution. The detection of misalignment between ownership and contribution is the first step towards investigating the impact it might have on the faster accumulation of Technical Debt.

6.1 Introduction

The development of software-intensive products and services has been evolving towards "componentising" architectures. Nowadays, big software development organisations tend to build their software products as a constellation of components often developed by different teams. One example of this trend is the wide adoption of the *microservices* architectural style [75, 156]

Alignment between architecture and organisation, therefore, plays a critical role [24, 156] in the development of large scale software systems. According to Conway's Law: Organisations that design systems tend to produce designs that mimic the communication structures of these organisations [53].

When we "componentise" (e.g., build the system using microservices), the team constellation should be adapted to minimise communication overhead. Although there are different approaches towards ownership and autonomy, software development organisations usually rely on weak ownership principle [74], which boils down to a given team owning a set of components (or microservices). Ideally, that team is the owner and responsible for that component and the main (or sole) contributor to the code. However, when examining real-world cases, that is seldom the case [32, 84]. In reality, a component can be providing support to several business streams; it can be developed by different teams and require the intervention of specialised teams to ensure non-functional requirements (e.g., security and reliability). Therefore, although the component will continue to be the responsibility of a team, the degree of contribution to the component can vary a lot.

The degree of alignment between the code ownership (i.e., the fact that the team is appointed as officially responsible for the quality of a given component or service [158]) and contribution (i.e., the extent to which that team is the main *contributor* for that particular product or service) can impact the effectiveness and efficiency of all the teams involved. We hypothesise that *the misalignment between team ownership and contribution can increase communication overhead and can make, for example, the code review time or the lead time to implement code changes longer.* We also hypothesise that *the misalignment may impact the pace of TD accumulation in services in which this misalignment is more acute.* In some cases, the *owner* team might only be holding the final responsibility of acting as gate (quality) keepers. In other words, the team members are not implementing the changes but only making sure that the code introduced in the codebase adheres to certain quality criteria and coding standards.

Let us suppose the owner team, after some time, loses control over the code of a component they own. In that case, it might also lose (partially) the ability to judge the appropriateness of the new code being committed, or just be cluttered by the number of changes and either not respond in time (longer lead time) or limit the extent of the code review. Therefore, properly analysing ownership and contribution seems crucial to properly "componentise" the architecture at scale effectively. However, we have started describing the disease, but we have not yet arrived at the actual symptoms. How can we adequately define contribution? Contribution can be in for of volume of code being authored, the number of commits, the code complexity, the number of created tickets, and other factors.

In this paper, we present a model that aims to distinguish between the team, which is the "official" or "formal" owner of a component, vs the team or teams which are the main contributors. The combination of the metrics and their visualisation enables the intuitive and easy interpretation of the state of contribution. While creating the model we have considered the research on both industrial (e.g., [32, 84]) and OSS (e.g., [72]) development communities. We have combined the metrics and methods they use to create our model to understand the alignment between ownership and contribution.

Goal: Analysing the software development projects to measure developer contributions to identify the alignment or misalignment between ownership and contribution. And how the alignment impacts technical debt.

Questions: How we can identify misalignment between ownership and contribution? Does the misalignment impact the growth of technical debt?

Metrics: Number of commits, code complexity, code churn [150], number of tickets (e.g., Jira), ticket complexity, of pull requests, and pull requests complexity.

Although, there are many resemblances with the code ownership in OSS development, we want to highlight that we are studying the role of *team* code ownership and its alignment with *team* contribution.

We present a case study (Section 6.2) that firstly led to the creation of a model to calculate the proportion of contribution to a component (Section 6.3) and secondly to validate the model (Section 6.4). We present the initial results (Section 6.5) and discuss their implications (Section 6.6). Finally, we present the related work (Section 6.7) and conclude the paper with our future work plan (Section 6.8).

6.2 Industrial case study

We conducted a case study as part of an ongoing collaboration with an industrial partner. This section describes the goals, sample, data collection, and analysis.

Goal

The case study had two goals. The first goal was to analyse the software development process to measure developer contributions to identify misalignment between ownership and contribution. Furthermore, we were interested in investigating whether the misalignment between ownership and contribution impacts the accumulation of TD. This analysis led to the creation of a model (details in Section 6.3) to assess the alignment between ownership and contribution. The second goal of the case study was to validate the developed model with the relevant stakeholders (details in Section 6.4).

Sample and Population

We conducted the case study with a large company that has chosen to remain anonymous. The company develops smart banking and financial solutions. The company is involved in a project profile collaboration with the research team and the components were selected based on the availability and convenience. The company wanted to improve their solutions/products and ways of working and hence was willing to participate in the study and learn from its results. The company employs agile practices and DevOps with autonomous teams working with practices like Scrum and Kanban. It uses a microservice architecture. Two teams A and B (with five and four developers respectively), were selected by convenience and availability. The teams worked on the main/legacy components

and faced the challenges related to faster accumulation of TD and resolving pull requests.

Design

The case study was conducted using bi-weekly workshop meetings with these two teams for about eight months (Mar. 2021 – Nov. 2021). The research team (the authors) and the product manager, the product owner, and a few developers (whose number ranged from 2-3 depending on availability and requirement of the meeting) always participated in the meetings. Each meeting was arranged for about 30 to 45 minutes. During the first 20 minutes, the research team presented their findings and conducted focus groups for the rest of the time. They asked questions to identify metrics to capture the proportion of contribution and the root causes of the misalignment between ownership and contribution. The feedback from the teams then derived the next meeting sessions. Till mid of Nov. 2021, the feedback of the focus groups led to the creation of the model whereas the last focus group session held in end of Nov. 2021 was used to validate the developed model.

Data Collection and Processing

Through the focus groups (held from Mar. until Nov. 2021), the research team gathered data regarding formal component ownership. The research team also collected data from product code, version control (git), and issue tickets (Jira) to calculate team contribution. The data was collected through APIs provided by the applications used by the company, e.g., BitBucket API and Jira API. The collected data was pre-processed to remove anomalies (incomplete, inconsistent data). Relevant metrics like size of systems (in LOC), number of commits, code complexity, code churn, number of tickets, ticket complexity, number of pull requests, and pull request complexity.

6.3 The OCAM Model

The purpose behind the Ownership and Contribution Alignment Model (OCAM) is to calculate the proportion of the contribution of teams to a component. Since the contribution can come from different sources, the model considers contributions from code production, issuing tickets, and code reviews, as suggested by Bass et al. [20, pp. 355-356], e.g., code, issue, and pull request complexity respectively. The model is created in an iterative process through a case

study (see Section 6.2) while consulting professional developers from a software development organisation.

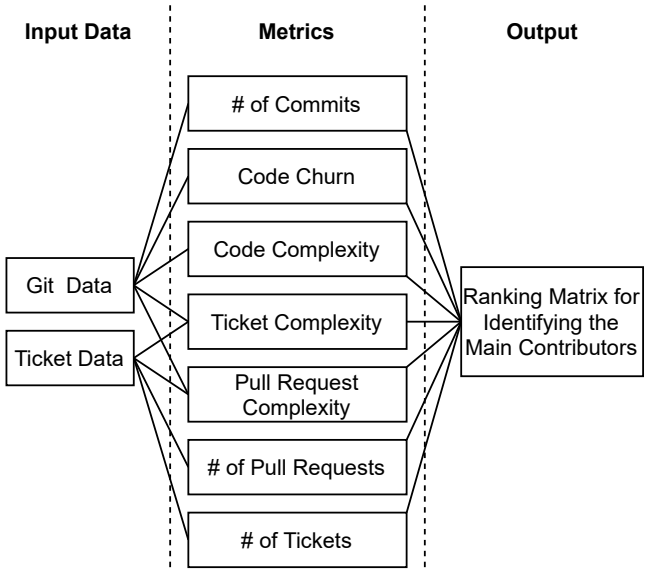

Figure 6.1: The Ownership and Contribution Model. The input data is fed to seven metrics and the model creates the contribution matrix.

The input data for the model is extracted from git and ticket system APIs. The model uses seven metrics to calculate the proportion of contribution for each metric for specified time duration and ranks the contributors. Finally, the model creates the contribution ranking matrix (see Fig. 6.1). The ranking matrix consists of metrics (rows) and teams (columns). A number is assigned to each cell in a row with the team's rank in that particular metric (row). A lower rank shows a higher proportion of contribution. These metrics are the number of commits, code churn, code complexity, number of tickets, ticket complexity, number of pull requests, and pull request complexity. There are no weights for the metrics in the model. The metrics can be calculated for individual developers or teams. The metrics are described in Table 6.1. The flexible nature of the model allows for removal of the metrics in case the data for calculating them are not available, i.e., any metric can be disregarded in the process in case of data limitation. Similarly, other metrics that can improve the quality of the model

can be included in the model. The final contribution matrix will be created based on the calculated metrics.

Table 6.1: The OCAM metrics and their descriptions. The metrics can be calculated for individual developers or teams.

Metric	Description
# of Commits	The total number of commits that were pushed to the repository in a period of time.
Code Churn [150]	The amount of code changed in a period of time, i.e., the ratio of written code in the codebase.
Code Complexity	Cyclomatic complexity [140] of the written code in a specific time.
# of Tickets	The total number of accepted tickets created in a period of time.
Ticket Complexity	The complexity of the accepted tickets created in a specific time. The complexity is calculated by examining the cyclomatic complexity of changes identified by `git diff` to the code in response to the tickets.
# of Pull Requests	The total number of pull requests created in a period of time.
Pull Request Complexity	The complexity of the pull requests created in a specific time. The complexity is calculated by examining the cyclomatic complexity of changes to the code identified by `git diff` in response to the pull requests.

Data was gathered continuously during the research and was presented to the team in every meeting. The purpose was to fine-tune the collection and interpretation of the data. Each metric in the model is calculated separately. The contributions are ranked for each metric. Once the metrics are collected, the contribution matrix is created and presented as a heatmap. Fig. 6.2 illustrates

an example output of the model and represents a snapshot of a component. By investigating the heatmap, we can understand the distribution of the contribution to a component. In this hypothetical example, four teams contribute to a component owned by Team C. However, the majority of the code is written by Team A. The misalignment of contribution and ownership is revealed by investigating the heatmap. Moreover, further details of the type of contribution can be investigated to shed light on the source of misalignment.

The output of the metrics are sorted and the team with the highest number has the most contribution. If two teams have the same number they are ranked equally. Lastly, in a case where two teams end up with similar contributions but different ranks for the metrics, their contributions remain the area of contribution. Such cases need to be individually investigated for each category of metrics or further for each individual metric.

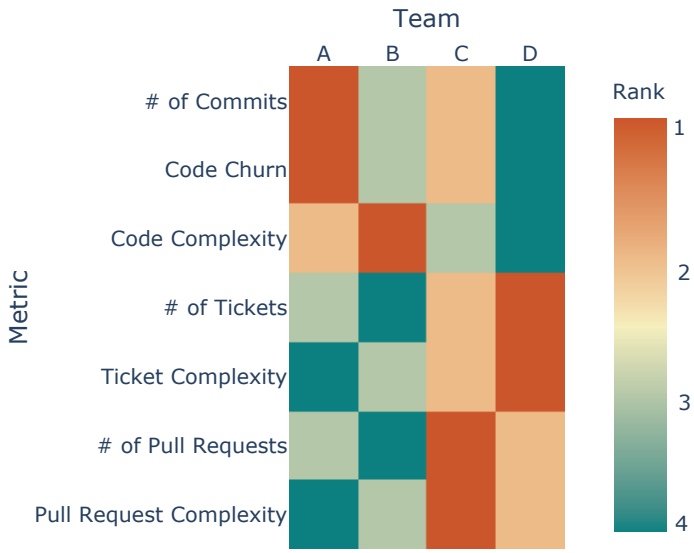

Figure 6.2: Example heatmap created by OCAM on a component developed by four teams. The component is owned by team C. 1:4 Most/Least Contribution.

6.4 Model Validation

After the model creation process, we assessed the model in practice with the help of the industrial partner. In particular, we are interested in testing whether the model can meaningfully identify the proportion of contributions to a component by answering the following research question.

> **RQ:** *To what extent can the proposed model correctly identify developer contributions to a component?*

We collected data from 267 components, developed by the company, as input for OCAM to investigate the alignment between ownership and contribution. We used the metrics provided in the model to create the ranking matrix for identifying the main contributors for each component. Lastly, we used SonarQube to calculate effort to repay TD (in minutes) for each component. We used SonarQube because it is widely used in both industrial and open-source systems [182] and has been used in other research studies, e.g., Zabardast et al. [230].

To validate the model, we designed a focus group session with participation of 2 product managers, the product owner, and four developers. The results of the analysis were presented and discussed among the participants to validate the model and its accuracy. During the focus group, the participants were asked 2 questions for each presented component. *1. Was the detection of misalignment correct or not?* and *2. What is the reason behind the misalignment?*

6.5 Initial Results

The model has identified 193 cases of misalignment, i.e., components where the owner team is not the main contributor. The final results of the analysis of 267 components were presented in a focus group with the participation of the authors and 2 product managers, 1 product owner, and 4 developers from teams A and B. 10 components from the output of the model were selected randomly to check for the validity of the model. The model correctly detected the misalignment in *all* the selected components. Each case was discussed separately between the researchers and participants.

Furthermore, we want to examine whether the misalignment between ownership and contribution impacts the accumulation of TD or not. We selected five components developed by two different teams (A and B). We extracted the amount of TD density per week during 2019 and 2020[4]. TD density is the total

[4]The company uses SonarQube in their development process.

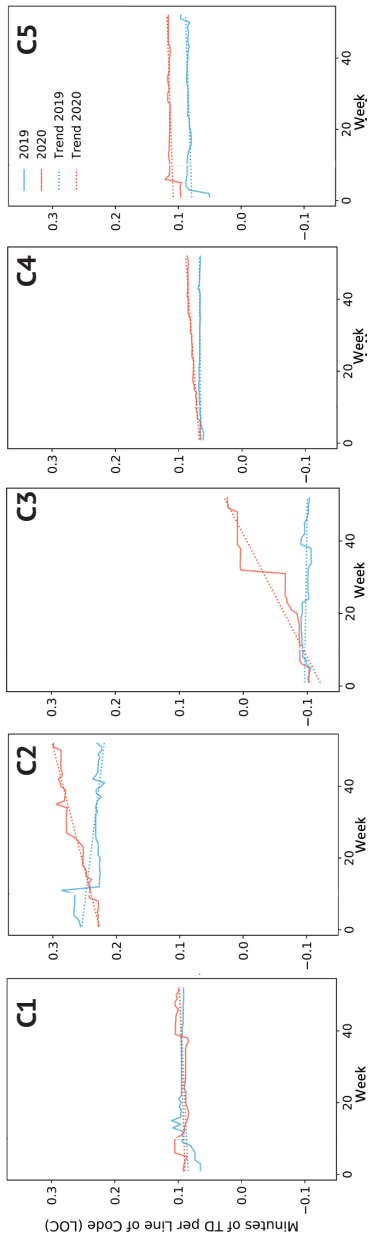

Figure 6.3: Technical debt density growth in 2019 and 2020 in five selected components.

amount of TD on a component normalised by its size [1, 59]. TD density allows us to compare the growth of TD with its relation to the growth of system size. TD density trends for these five components are presented in Fig. 6.3. TD density in components (C2) and (C3) increases much faster than the rest. The OCAM identified misalignment between ownership and contribution for components (C2) and (C3), which turned out to have a different pattern when it comes to the accumulation of TD. The results of this analysis were presented to teams.

6.6 Discussion and Practical Implications

OCAM provides an objective way to assess the contribution of developers from various perspectives. The results can be used in an informative way, i.e., to clarify the state of ownership and contribution to the developing teams. The results can help the developing teams approach development tasks more efficiently. Raising awareness regarding misalignment between ownership and contribution can help team understand why TD is increasing faster in some components, and therefore increase the level of quality required on code that is merged, or even find an explanation on why they are cluttered by the number of Pull Requests or bug reports they receive. The initial results presented in this paper are the first steps to investigate such factors further. We postulate that early identification of such misalignment can help organisations act in time to prevent faster accumulation of TD. Moreover, the model provides valuable insights for managers when investigating problems that stem from the organisational structure.

6.6.1 Limitations and Threats to Validity

Construct Validity We use different tools for measurements that reflect the construct of our study. We are aware of the limitations imposed by the tools we used in our study. We have selected widely-adopted, industrial *de-facto* standard tools, such as SonarQube, to measure the metrics used for this study. Ownership and contribution alignment is a complicated and multifaceted phenomenon to study and there are many dimensions to consider while studying it. While OCAM presents metrics from different categories, there might be other metrics not considered on this study, or the ones being considered might not reflect the ownership vs contribution alignment. We created the model to be extensible to mitigate the model's limitations.

Internal Validity As stated above, the problem under study is complex, and there might be other factors that might explain some of the findings. We have used a methodological triangulation [181] to mitigate those threats, using quantitative and qualitative sources. We continue working on better understanding these factors to minimise their impact on the results.

External Validity Another threat to the study presented in this paper is the generalisability of the results into different contexts out of the investigated cases. We are aware that our results are strictly applicable to the studied case. In this study, we are focusing on analytical generalisability. The main goal is to identify potential signals for misalignment between ownership and contribution.

Reliability Reliability is concerned with the data and the analysis being independent of the specific researchers. This is the most significant threat to the validity of our study. By including the company in focus group sessions for interpreting the results, we have tried to mitigate this threat.

6.7 Related Work

Developer Contribution Metrics

There are many metrics derived from version control systems such as git that are used to calculate the amount of developer contribution. de Bassi et al. [57] provide a collection of code quality metrics. The authors categorise them in four groups of *Complexity Metrics*, *Inheritance Metrics*, *Size Metrics*, and *Coupling Metrics*. They successfully evaluate the individual contributions using quality metrics. Similarly, Diamantopoulos et al. [58] analyse the contribution of project collaborators in 3000 most popular Java projects on GitHub. The authors investigate 19 metrics in 2 categories of *Development* and *Operations*, e.g., "commits authored" and "issues participated" respectively for each category. Parizi et al. [168], present a git-driven solution to measure team members' contribution during the development. They propose five metrics including *number of commits*, *number of merge pull requests*, *number of files*, *total lines of code*, and *time spent* to measure contribution. The provided metrics by the authors do not consider the difficulty of the project. However, the authors note that the difficulty should always be considered when evaluating the contribution of developers. Lastly, Oliveira et al. [161] classify developer productivity into *code-based metrics* and *commit-based metrics*, e.g., "code owned by time" and "commits/time" respectively for each category. The authors have interviewed two organisations to understand the perception and relevance of the metrics

in practice. Their results suggest that there is a "positive impression" for the adoption of code-based metrics in the organisations. The studies related to capturing the developer contributions use similar code-metrics and categories to identify the proportion of contributions from developers.

Architecture and Organisation

The alignment between architecture (technical dependencies) and organisation (organisational dependencies), i.e., Conway's law [53], has been studied over the years. There are several studies that investigate socio-technical congruence using different metrics, e.g., [95, 139]. In an ideal world, the owners of a component should be solely responsible for developing it. Misalignment between ownership and contribution might negatively impact the efficiency of development [95], causing other problems in the architecture, the organisation, or both. Architectural problems can manifest as architectural technical debt (ATD). In order to capture ATD, Bass et al. [20, pp. 355-356] suggest using three types information including "Source Code", "Revision History", and "Issue Information". Therefore, misalignment between architecture and organisation can be a sign of ATD and a model to identify such misalignment can be helpful.

6.8 Conclusions

This study aims to create a model to measure the degree of contribution to components. The contribution is a multi-dimensional phenomenon that cannot be represented by individual metrics. Our model (OCAM) for measuring the degree of contribution to components uses existing metrics collected during development without burdening organisations with the installation of extra tools. The contribution can be assessed from three main perspectives of written code, issued tickets, and code reviews.

Our initial analysis of the case study on 267 components revealed 193 cases with a misalignment between ownership and contribution. We conducted a focus group session (end of Nov. 2021) with the developing teams to validate the model's findings. We aim to expand the work introduced in this short paper, first by refining the model and evaluating more cases using OCAM. In doing so, we plan to improve the model based on the gathered evidence. Lastly, our goal is to release the model's source code and implement the final version of the model in practice report on our experience.

Chapter 7

The Impact of Forced Working-From-Home on Code Technical Debt: An Industrial Case Study

This chapter is based on the following paper:

> **Ehsan Zabardast**, Javier Gonzalez-Huerta, and Francis Palma. "The Impact of Forced Working-From-Home on Code Technical Debt: An Industrial Case Study" *In 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (pp. 30-34). IEEE (2022).[1][2][3]

[1] *© 2022 IEEE. Reprinted, with permission, from Ehsan Zabardast, Javier Gonzalez-Huerta, and Francis Palma "The Impact of Forced Working-From-Home on Code Technical Debt: An Industrial Case Study", 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), 2022*

[2] https://doi.org/10.1109/SEAA56994.2022.00054

[3] *In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of BTHs products or services. Internal or personal use of this material is permitted.*

Background: The COVID-19 outbreak interrupted regular activities for over a year in many countries and resulted in a radical change in ways of working for software development companies, i.e., most software development companies switched to a forced Working-From-Home (WFH) mode.

Aim: Although several studies have analysed different aspects of forced WFH mode, it is unknown whether and to what extent WFH impacted the accumulation of technical debt (TD) when developers have different ways to coordinate and communicate with peers.

Method: Using the year 2019 as a baseline, we carried out an industrial case study to analyse the evolution of TD in five components that are part of a large project while WFH. As part of the data collection, we carried out a focus group with developers to explain the different patterns observed from the quantitative data analysis.

Results: TD accumulated at a slower pace during WFH as compared with the working-from-office period in four components out of five. These differences were found to be statistically significant. Through a focus group, we have identified different factors that might explain the changes in TD accumulation. One of these factors is responsibility diffusion which seems to explain why TD grows faster during the WFH period in one of the components.

Conclusion: The results suggest that when the ways of working change, the change between working from office and working from home does not result in an increased accumulation of TD.

Keywords: *Technical Debt, Empirical Study, Industrial Study, Case Study, COVID-19, Telework, Work From Home*

7.1 Introduction

Technical Debt (TD) is a metaphor to explain the long-term consequences of sub-optimal decisions taken to give priority to speed on deliveries [55] and deals primarily with non-visible aspects and issues of software development, and its maintenance [106]. Technical and design decisions influence the introduction of TD. However, we should not forget that those decisions are taken by humans (e.g., developers, architects, testers) in their daily work. The environment and the conditions in which the work is done can significantly impact whether or not we take the best solution at hand and, therefore, whether we incur TD. The communication and coordination among members within and outside the team

in the organisation might be a factor that influences how developers incur and repay TD [21]. These working conditions might affect how and when we take actions to mitigate and payback TD.

In March 2020, with the COVID-19 pandemic, software development organisations had to confront a challenging situation: how to continue their activity working 100% on distance, with their employees working from home (WFH). This new way of working is different from distributed software development, in which teams are located in different sites. Therefore, in distributed software development, intra-team communication happens *face-to-face*. In contrast, inter-team communication usually should be mediated using instant messaging and video conference tools, while in WFH, even intra-team communication should be mediated with online tools. This sudden change exposed organisations to many challenges, uncertainties, and risks. There have been studies to analyse changes in the working routines of developers [17, 71, 173, 193]. However, very little is known about the influence that WFH had on TD. Some companies suggested a change in their working strategy during the pandemic, enabling their employees to work from distance. Therefore, it is important to understand how software artefacts degrade considering different ways of working.

In this paper, we present the results of an industrial case study to analyse how the introduction and repayment of TD have fluctuated during the first nine months of *WFH* and compare it to the same interval working from *office*, by studying several repositories from a software development organisation. The case study comprises two different data collection methods. On the one hand, we collected quantitative data utilising archival analysis. On the other hand, we carried out a focus group with the development team to collect qualitative data. The goal of this mixed-method is to triangulate data to strengthen the evidence and identify the factors that impact TD. We address the following research question:

> **RQ:** *How the forced Working-From-Home mode impacted the accumulation of technical debt?*

More specifically, whether and to what extent WFH impacted the accumulation of TD when developers have different ways to coordinate and communicate with peers.

The remainder of this paper is structured as follows. In Section 7.2, we present the design of the study while Section 7.3 presents the results. Section 7.4 discusses the main findings and Section 7.5 summarises the threats to validity. In Section 7.6, we discuss related work in the area. Finally, Section 7.7 draws our conclusions and discusses further work.

7.2 Research Methodology

To address the research question, we carried out an industrial case study that combines archival and qualitative data collected from `git` repositories, SonarQube, and a focus group. Fig 7.1 shows our research methodology.

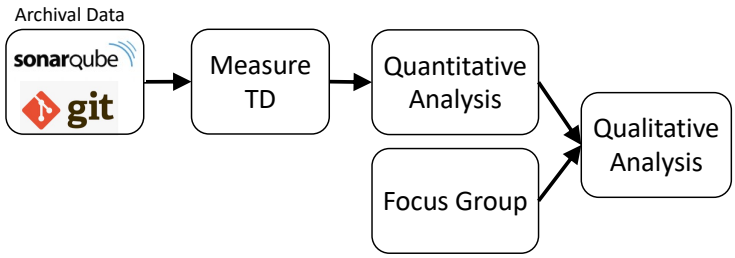

Figure 7.1: The research methodology.

This study is conducted in an industrial setting and comprises the analysis of five components developed by a company that has chosen to remain anonymous. The company is a large software development company that develops web-based financial and accountancy services. It is a mature company in its development practices and has well-established, successful products. The case has been selected by convenience (availability and access). The company is interested in continuously improving its products and ways of working. Therefore, it was willing to participate in the study and learn from its results.

We are presenting an embedded case study [225] where we are analysing five components (units of analysis), which are mainly written in `Java`. The analysis period of all components spans two years (2019 and 2020).

In the following, we describe the main constructs and measurements used for the quantitative analysis in this research, as illustrated in Fig 7.2. We calculated the amount of accumulated TD per week for each component. We used SonarQube[4] to calculate the amount of TD in each component and calculated the amount of code using gitlog. The company uses SonarQube for controlling code quality and as a proxy for TD in the development process. Moreover, SonarQube has been used in similar studies on the topic of TD, e.g., [60, 230].

[4]Version 8.9.6 LTS at https://www.sonarqube.org

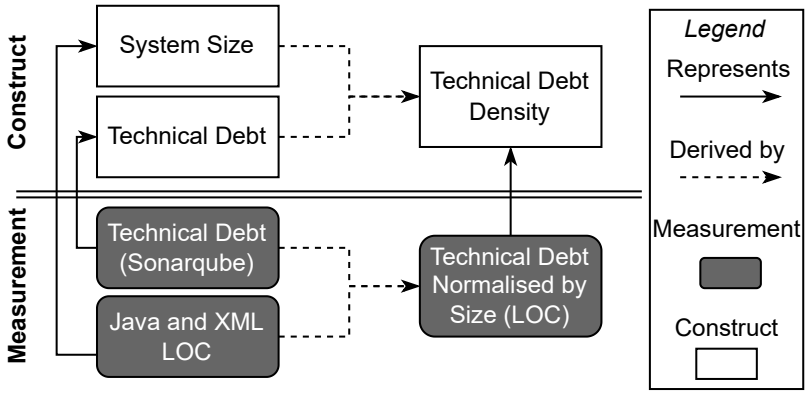

Figure 7.2: Study constructs and measurements.

7.2.1 Data Collection

Quantitative Data

The data gathered from SonarQube is the issues detected by the quality profile and gate[5] selected by the company. The issues are representative of the quality profile and quality gate. The data is collected via the company's SonarQube API, i.e., the company uses the analysis of SonarQube as a proxy for the components TD. The tool provides the estimated time (i.e., effort in minutes) required to resolve the issues. The accumulated TD of a component is the total remediation time for all the issues detected. We considered repaid TD when issues are tagged as 'fixed' and 'closed'.

To calculate TD density, we used the growth in the size of the system reported by git log, as added lines of code minus deleted lines of code. Table 7.1 presents the details of the investigated components in this study.

Qualitative Data

The results of the quantitative analysis were presented to the development team (see Table 7.2) responsible for the development of the five components in a focus group session. The developers provided descriptions and explanations for what they thought were the reasons behind some of the most prominent

[5]https://docs.sonarqube.org/latest/user-guide/quality-gates/

Table 7.1: Investigated components' size and number of commits.

ID	Size (KLOC)	# commits in 2019	# commits in 2020
C1	6.2	314	292
C2	9	315	362
C3	4.8	163	352
C4	14	404	269
C5	6.5	149	499
Total	40.5	1,345	1,774

changes in TD or size during the analysed period, and their experience with factors that impacted the accumulation of TD. The statements were collected via sticky notes and the transcript of the focus group session. The length of the focus group was 60 minutes, and six members (out of a total of 8 in the team) participated in the session. In order to complement the statements, we recorded and transcribed the conversation during the focus group. The focus group included the following steps:

Table 7.2: Focus group participant information.

	Experience (years)	
Role	In Company	Overall
Product Owner	8	21
Development Manager	2	18
Product Owner	4	9
Senior Developer	12	12
Senior Developer	2	6
Scrum Master	5	5

- Participants introduced themselves and their roles in the team.
- The authors of this paper presented a summary of the results and explained the focus group procedure.
- Participants answered questions regarding the results of the quantitative analysis.

  - Participants read each question. They had five minutes to think about each question. They wrote their answers and opinion on sticky notes and posted them at the end of the time altogether.

- Participants and researchers had a closing discussion where they discussed the details of the results.

7.2.2 Data Analysis

The data analysis is divided into two sections where we analyse the quantitative and qualitative data separately.

Quantitative Data Analysis

To evaluate the state of each component, we use $TD_{density}$. $TD_{density}$ is the normalised amount of TD per line of code [1, 59]. Plotting $TD_{density}$ over time will allow us to visualise the changes in TD density and their trend. The weeks from 2019-01-01 to 2020-03-11 were considered as *office* and the weeks from 2020-03-11 to 2020-12-31 were considered as WFH. The vertical black line is used to mark the week in which the company switched to forced WFH (i.e., it only applies to the year 2020).

For comparing whether there were differences in the TD density while working in the office vs WFH, we use the t-test for independent samples (at the significance of 0.05) in case the samples were normally distributed or the Mann-Whitney tests in case the samples were not normally distributed [69]. For assessing whether each sample (i.e., Office and WFH) were normally distributed, we used the Shapiro-Wilk Test [69]. The tests were conducted to compare the same interval (i.e., March to December) between *office* and *WFH* periods, although Figure 7.3 shows the two natural years.

Qualitative Data Analysis

The main input for the qualitative data analysis is the statements provided by the focus group participants. We have used *In Vivo Coding* to label the statements. In Vivo coding is suitable for labelling raw data in the first cycle coding. It prioritises the opinions of the interviewees [184]. The coding was done by two authors independently. Then we compared all the labels and discussed the conflicting cases with the help of the third author.

For the second cycle coding, we have used *Pattern Coding* to create the themes from the labels from the previous step. Pattern coding allows us to

identify the themes from the data [184]. One of the authors extracted the emerging themes and later discussed them with all authors to agree on themes.

7.3 Results

7.3.1 Results from the Quantitative Data Analysis

Fig 7.3 illustrates the data for the quantitative data analysis, in which each row depicts a different component. The *blue* colour is used for year 2019 and *red* for year 2020. The vertical *black* line is used to mark the week in which the company switched to forced WFH—in 2020. The horizontal scale on each graph represents the weeks ranging from 1 to 52. For each component, we present the results using four different graphical representations (four columns). These columns are:

- **Column I: TD per Week** shows the amount of TD in a component per week. The positive and negative values represent introduced and repaid TD, respectively.
- **Column II: Accumulated TD** shows the accumulated TD during each year. The dashed blue and red lines are trends for the years 2019 and 2020, respectively, and have been calculated using the linear regression function of Scipy package[6], version 1.6.1 for Python 3.8.
- **Column III: Component Size Growth** shows the change in the component size during each year. The size is calculated only considering `Java` and `XML` files since the quality gate only considers issues related to `Java` and `XML`.
- **Column IV: TD Density** shows TD normalised per LOC during each year. The dashed blue and red lines are trends for the years 2019 and 2020, respectively, and have been calculated using the linear regression function of the Scipy package version 1.6.1 for Python 3.8. The trend lines in Column IV are all significant ($p - value < 0.5$). For **C1, C2, C3** and **C5**, the trend lines explain $< 50\%$ of the variance ($R^2 < 50\%$). However, for $C4_{WFH}$, the trend line is higher and explains 75% of the variance ($R^2 = 74.86\%$) and for $C4_{office}$ the trend line is slightly lower than 50% ($R^2 = 43.89\%$).

[6] https://www.scipy.org

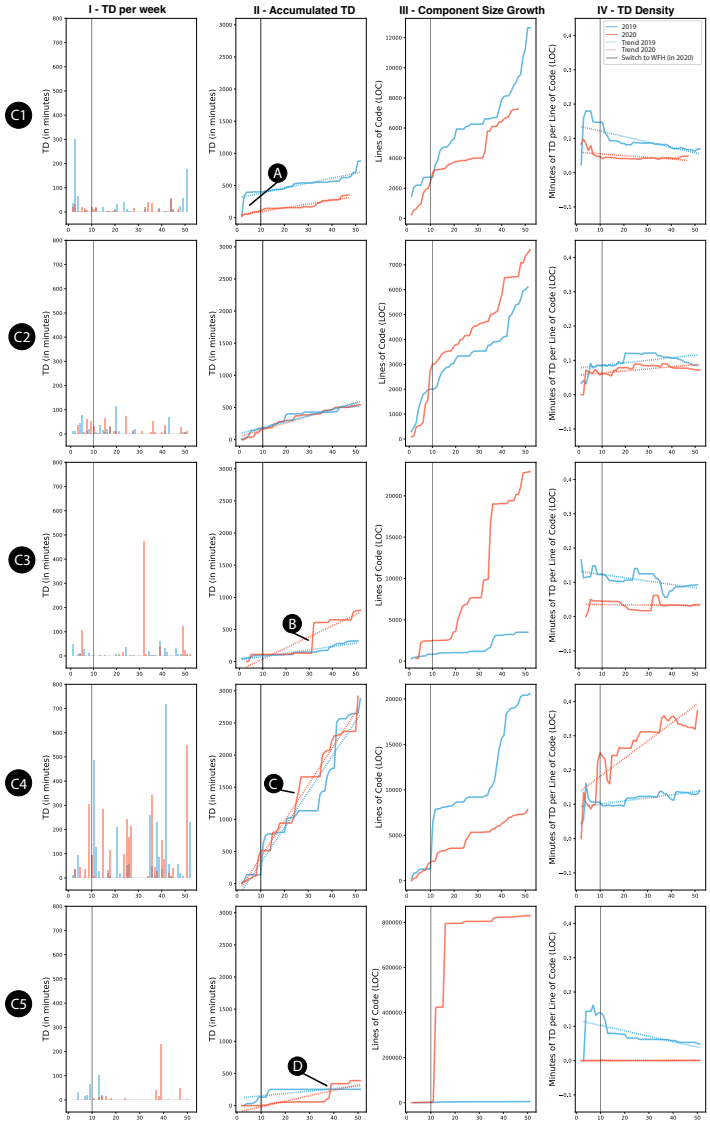

Figure 7.3: Results of the quantitative analysis for five components. Column I shows TD per week; Column II shows the accumulated TD; Column III shows the component size growths; and Column IV shows the TD density. The vertical black line is used to mark the week in which the company switched to forced WFH (i.e., it only applies to the year 2020).

We observe statistically significant differences in the changes of $TD_{density}$ in all components when comparing 2019 and 2020. We observe that, when working from office, components **C1**, **C2**, **C3**, and **C5** incur more TD, i.e., on average $TD_{density}$ is higher during 2019. Therefore, the accumulated $TD_{density}$ introduced since the beginning of the year 2019 is higher than the year 2020 during the WFH period. Conversely, component **C4** incurs more TD when working from home, i.e., on average $TD_{density}$ is higher during 2020. Therefore, the accumulated $TD_{density}$ introduced since the beginning of the year 2019 is lower than the year 2020 during the WFH period.

Table 7.3: Descriptive statistics for each component for TD density.

Component	N	Min.	Max.	Mean	Median
$C1_{office}$	51	0.02	0.18	0.09	0.09
$C1_{WFH}$	46	0.04	0.1	0.05	0.04
$C2_{office}$	51	0.03	0.12	0.1	0.1
$C2_{WFH}$	52	0	0.09	0.07	0.08
$C3_{office}$	51	0.06	0.17	0.11	0.11
$C3_{WFH}$	50	0	0.06	0.03	0.03
$C4_{office}$	51	0.04	0.16	0.12	0.12
$C4_{WFH}$	50	0	0.37	0.27	0.3
$C5_{office}$	49	0	0.16	0.08	0.06
$C5_{WFH}$	50	0	0	0	0

Table 7.3 summarises the descriptive statistics for TD density per week. We applied the Mann-Whitney test, provided that the samples were not normally distributed. The results of the Mann-Whitney test results are as follows:

- **C1.** There were *significant* differences in the introduced $TD_{density}$, which grew faster when working from office as compared to WFH ($W = 2203$, p=0.000).
- **C2.** There were *significant* differences in the introduced $TD_{density}$, which grew faster when working from office as compared to WFH ($W = 2307$, p=0.000).
- **C3.** There were *significant* differences in the introduced $TD_{density}$, which grew faster when working from office as compared to WFH ($W = 2544$, p=0.000).

- **C4.** There were *significant* differences in the introduced $TD_{density}$, which grew faster WFH as compared to working from office ($W = 263$, p=0.000).
- **C5.** There were *significant* differences in the introduced $TD_{density}$, which grew faster when working from office as compared to WFH ($W = 2404.5$, p=0.000).

7.3.2 Results from the Qualitative Data Analysis

In the focus group, we summarised the results presented in Section 7.3.1 to the developers, and asked them to elaborate on the results. We asked the participants to provide all the factors that impacted TD to help us understand and study the impact of those factors within the context of WFH. We recorded the number of participants who agreed on each statement (ratio of agreement in Fig 7.4). Example statements are provided.

- ***Question:*** *"Is there any explanation for the change of TD in each component or on the team's ways of working?"* The statements from developers suggest that there are several different reasons for the accumulation of TD in each component. The summary of possible explanations for the accumulation of TD in the investigated components are (ordered by level of agreement among participants):
  - *Changes in mob programming*[7] *(5.6/6):* The development team engaged in less mob programming. The developers state that they changed to individual programming, which is not as effective while working from home, and it impacts the accumulation of TD. *"a change from working in a mob towards working more individually"*
  - *Growth in the complexity (5/6):* Growth in complexity of the component makes TD growth faster. *"company growth, new products and projects"*
  - *Responsibility diffusion (5/6):* Having many external contributors to the repository, i.e., responsibility diffusion, impacts the accumulation of TD. *"[the component] has a lot of contributors outside our team"*
  - *Switch to another framework (4/6):* Migration to a new framework impacts the accumulation of TD. *"switch to [a new framework] lead to many changes over many components"*

[7] Mob Programming is when the whole team works on the same thing, at the same time, in the same space, and on the same computer.

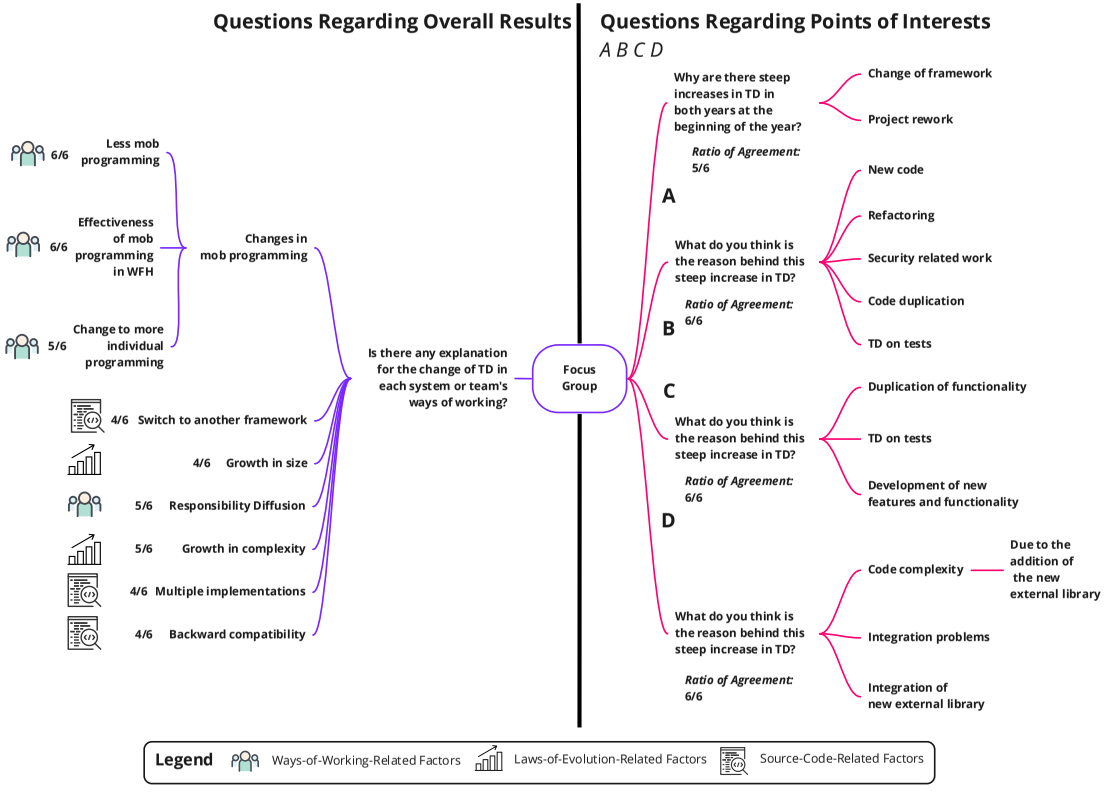

Figure 7.4: Summary of the results of the qualitative analysis. The number next to the factors represent the ratio of agreement among the participants.

  - *Growth in size (4/6):* Growth in the component via new development impacts the accumulation of TD. *"2020 seems to have had larger changes"*
  - *Multiple implementations (4/6):* Multiple implementations of (a part of) code base impact the accumulation of TD. *"multiple implementations [...] may increases TD"*
  - *Backward compatibility (4/6):* Backward compatibility might impact the accumulation of TD. *"backward compatibility may increase TD"*

There are points of interest in some of the charts presented in Fig 7.3, e.g., in component **C1**, Column II marked with *A* which is a significant increase in the introduction of TD at the beginning of both 2019 and 2020. To elaborate on such points of interest, we asked developers specific questions regarding such observations. These are four observations marked as *A, B, C,* and *D* in Fig 7.3.

- ***A: "Why are there steep increases in TD in both years at the beginning of the year?"*** The developers stated that this steep increase in TD correlates to the migration to a new framework and duplication of work because of "*project rework*". Ratio of agreement in the focus group for this answer was 5/6.
- ***"What do you think is the reason behind this steep increase in TD?"***
  - **B:** The developers state that this steep increase in TD correlates to the introduction of new code, refactoring, security maintenance work, code duplication, and TD on tests. Ratio of agreement in the focus group for this answer was 6/6.
  - **C:** The developers state that this steep increase in TD correlates to the duplication of functionality, TD on tests, development of new features and functionality, and refactoring. Ratio of agreement in the focus group for this answer was 6/6.
  - **D:** The developers state that this steep increase in TD correlates to the integration of a new external library to an already complex code. Ratio of agreement in the focus group for this answer was 6/6.

The statements provided by the participants were collected via sticky notes. We analysed the statement to extract labels and themes, as described in Section 7.2.2. We extracted three main themes of factors that the participants in

the focus group perceive might have an impact on the growth of TD including: source-code-related factors, laws-of-evolution-related factors, and ways-of-working-related factors. The results of the thematic analysis are summarised in Fig 7.4.

The *source-code*-related factors include code, architecture, and design decisions that impact the growth of TD such as *change of framework*, *multiple implementation of a component*, and *backward compatibility of a component*. The *laws-of-evolution*-related factors include continuous change and increasing complexity [116] that impact the growth of TD, such as *growth in size* and *growth in complexity*. The ways-of-working related factors include communication and coordination of the team that impact the growth of TD, such as *changes in mob programming* and *contributions of developers outside of the team*.

Regarding *ways-of-working*-related factors, the participants in the focus group pointed out that the component **C4** does not have a clear ownership. The team that formally owns the component is not among the main contributors in terms of the number of revisions or the volume of code being contributed to the component (i.e., when analysing the whole history of the component, the *owning* team was the fourth in the number of total revisions). The *owning team* acts as *"gate keepers"* (i.e., in their own words) in the sense that members of this team are responsible for rating the pull requests, but the team does not have full control over the code being included in the component. This is a clear case of *responsibility diffusion* [209], where many other teams are contributing to a component, and no one takes the required code maintenance actions to keep the code in shape and/or repay TD. We observe that, apparently, in component **C4**, the effect of responsibility diffusion on TD seems to be exacerbated during the WFH period, with less pair or mob programming and more difficult inter-team communication.

7.4 Discussion

As highlighted in Section 7.3, the week-by-week statistical analysis show statistically significant differences. TD is accumulating at a faster pace in the component **C4** during the forced WFH period while in the other components TD seems to accumulate faster when working from office. The developers expected the decrease in TD accumulation: *"A gut feeling is that technical debt should not have increased during WFH - rather decreased. Sharing screens and discussing feels more natural when working remotely."* Our results show that

the practices used by the developers during WFH such as screen sharing and impromptu meetings have helped decrease the accumulation of TD.

We investigated the faster increase of TD growth in component C4. Further investigation of the contributions revealed that developers from other teams are the main contributors to this component. There are 77 developers from 17 teams contributing component **C4**. The impact of responsibility diffusion on the growth of TD is highlighted as a major factor by the developers. Tornhill [209] suggests that the developers who do not own the *code* tend to disregard "*responsibility for the quality and future of a piece of code*" which is aligned with our observations and results. The gatekeeping solution can also, in the long run, have negative consequences for the system due to review fatigue [209].

Overall, the results suggest that even when the ways of working change, e.g., less mob programming, the change between working from office and working from home does not result in increasing accumulation of TD. However, in cases where there are many different teams contributing to a component and there is *no clear ownership* of the component, the change from working from office to working from home may result in an increase pace in the accumulation of TD.

The results of the qualitative data analysis show that different factors can impact the accumulation of TD. These potential factors can be divided into *source-code*, *laws-of-evolution*, and *ways-of-working*-related factors. While we expect all the potential factors to impact TD, we observe that the impact of ways-of-working related factors can be exacerbated by forced WFH mode in the components with *less clear ownership*, as stated by the participants in the focus group.

7.5 Threats to Validity

In this section, we discuss the main threats to validity from four different perspectives [225]: construct validity, internal validity, external validity, and reliability.

Construct validity is the degree to which the measures studied can reflect the constructs that the researcher is aiming at and have been defined in the research questions [174]. We use SonarQube, which is a widely-adopted, *de-facto* standard used in industry to search for TD [135], and that has been used in similar other research studies (e.g., [59, 230]). We also rely on a metric $TD_{Density}$ [1], that has been used to monitor the impact of external factors (e.g., Clean Code) in [60]. The calculation of $TD_{Density}$ also takes the component's size as input, and for its calculation, we considered `Java` (source code and

tests) and `XML` files (integration tests). All TD items have the same weight in our analysis and they are considered equally important. To mitigate potential threats regarding the size calculations, the selection of the file types was made in consensus with the heads of development in the studied company. The repaid TD is calculated through the issues flagged as 'fixed' and 'closed'. However, there might be cases where TD has been removed by deleting the files that might not be included in our analysis. Finally, we applied a set of commonly used statistical tests employed in the empirical SE community that relate to the constructs described in Figure 7.2.

Internal validity concerns whether there might be other factors that might explain some of the findings (e.g., the company is migrating to use a new framework which might impact the amount of accumulated TD). We have used a methodological triangulation [181], to try to mitigate those threats, using quantitative and qualitative sources. From the qualitative data, we understood that, for example, the lack of formal ownership and the usage of a quality gate might explain the uncontrolled growth of TD in the WFH period. However, there might be other reasons, and further studies are needed to understand better additional factors that can explain this exacerbated growth. There might be other non-studied factors that can explain the differences in TD during the studied periods.

External validity concerns the generalisability of the results into different contexts out of the investigated cases. Our results are strictly applicable to the studied case. One of the common misunderstandings about case study research is the inability to generalise from single cases [70]. While interpreting the results, we do not focus on statistical generalisation. However, we focus on comparing different components and searching for plausible explanations, trying to extract themes that can explain the observed differences.

Reliability concerns whether the data and the analysis are independent of the specific researchers. This might be a significant threat to the validity of this study. We tried to mitigate this threat by first involving the studied company in interpreting the results instead of restricting the analysis to a pure comparison of two data frames. Second, we tried to improve the reliability of the qualitative analysis by doing a separate blinded coding.

7.6 Related Work

This section summarises studies from findings on WFH regarding software development and productivity, working patterns and TD, and measuring the TD trend.

Studies Analysing the Impact of WFH: One of the earliest studies during the COVID-19 pandemic by Bao et al. [17] analysed developers' activity records to understand the impact of WFH on productivity. The interest group is 139 developers in Baidu Inc, China. The dataset is obtained between Dec. 2019 and Mar. 2020 and compared with the corresponding months in 2018 and 2019. However, no generalised observation could be made, i.e., although WFH has different impacts on developers' productivity, this varies by the project age, type, language, and size. Also, the productivity of the majority of the developers when WFH is similar to the office. Smite et al. [193] studied work patterns for a geographically distributed software development company to understand how employees cope with the WFH mode. Findings suggested that during the WFH, engineers follow daily routines similar to onsite work. Moreover, there is no significant change in code production comparing the onsite and WFH code commits. The authors reported an increased flow and productivity in some instances (e.g., familiarity with the tasks) due to a lack of spontaneous interruptions from colleagues and shorter breaks.

Working Patterns and TD: In an industrial case study, Codabux and Williams [50] reported that developers primarily focus on design, testing, and defect-related TD in an agile setting. In the agile and distributed environment, developers often create their taxonomy of TD subject to the type of tasks they were assigned and their understanding. Yli-Huumo et al. [226] provided an exploratory, qualitative case study interviewing 25 engineers in eight development teams and analysing empirical data to understand TD management. TD management was done systematically for most of the teams working in a distributed environment, i.e., managers reserved 20% of the development time to improve the code base and refactor TD issues.

Measuring the TD Trend: When it comes to analysing TD in a project, there are tools (e.g., SonarQube) that report the number of TD items. These tools can use that information to estimate the total time to remediate them, approximating the TD principal accumulated in the project. However, using this information to assess the TD evolution over time might be problematic, especially if we want to establish comparisons between two different periods or

compare evolution across different systems. If we compare two periods of time in which the development activity resulted in the volume of code being added to the systems differs substantially, analysing only the accumulated TD principal might lead us to the wrong conclusions.

The density of TD or $TD_{density}$ [1, 59] defined as the TD normalised per line of code, can allow us to reason about the evolution and the impact that external factors might have on the accumulation of TD [1, 59]. In [59] analyses the impact that clean code might have on TD. In this particular case, $TD_{density}$ reflects how the TD density tends to decrease when the newly added code is cleaner. In contrast, the deletion of high-quality code causes an increase in TD density. Thus, the system's maintainability during evolution might be strongly associated with the $TD_{density}$ [59].

With forced WFH mode and the newly adopted work from anywhere, the state of TD needs to be investigated in-depth. The ways of working impact TD, e.g., whether developers are working remotely or at the designated work-space co-located with their peers [21]. Using $TD_{density}$ can allow us to draw better conclusions regarding its potential impact.

7.7 Conclusion

TD reflects the long-term consequences of sub-optimal decisions taken to accomplish short-term design and development goals [55]. Due to the global COVID-19 outbreak, developers are forced to work from home worldwide. We investigated the impact of forced WFH on the accumulation of TD. We showed the accumulation of the TD in terms of TD density.

We analysed five components to study the impact of forced WFH on the accumulation of TD in an industrial case. While comparing the accumulation of TD between the pre-pandemic (in 2019) and pandemic (in 2020) times, the tests showed statistically significant increase in the accumulation of TD in one component identified with lack of ownership and responsibility diffusion.

Based on the focus group, we identified that factors related to *source-code*, *laws-of-evolution*, and *ways-of-working* are among the factors that impact the growth of TD. We plan to extend and replicate this study on other components and in other organisations, which may strengthen the evidence regarding the impact of WFH on the accumulation of TD. We want to examine other confounding factors (e.g., size of the systems, developers' experience, working conditions at home, and delivery pressure) in relation to forced WFH and TD

accumulation. Finally, we plan to explore how misalignment between ownership and contribution impacts TD.

Chapter 8

The Impact of Ownership and Contribution Alignment on Code Technical Debt Accumulation

This chapter is based on the following paper:

> **Ehsan Zabardast**, Javier Gonzalez-Huerta, Francis Palma, and Panagiota Chatzipetrou. "The Impact of Ownership and Contribution Alignment on Code Technical Debt Accumulation" *Submitted to Transactions on Software Engineering.* (Under Review) IEEE (2023)[1]

[1]Preprint available on: https://arxiv.org/abs/2304.02140

Context: Software development organisations strive to maintain their effectiveness whilst the complexity of the systems they develop continues to grow. To tackle this challenge, software development organisations tend to be organised into small teams working with components that can be developed, tested, and deployed separately. In this scenario, organisations must design their software architecture and organisational structures in such a way that enables communication and minimises dependencies, as well as helps teams reduce code and architectural degradation. Ensuring that each small, independent team is responsible for the components they primarily contribute is one approach to achieving this goal.

Objective: This article reports a study that aims at understanding the impact of ownership and contribution alignment (contribution alignment, for short) on accumulation of code technical debt (TD) and how abrupt changes in team constellation affect teams' effectiveness in managing TD.

Method: We have conducted an embedded case study in a software development company developing a very large software system, analysing ten components belonging to one team. During the studied period, the team was split into two, and the components owned by that team were distributed between the two new teams. We have collected archival data from the company's tools in their daily development operations.

Results: In most cases with high degrees of contribution alignment, we have noticed a negative correlation between contribution alignment and TD per line of code (TD Density) before the team split. In four components, this correlation is statistically significant. This means that a higher contribution alignment degree implies a lower TD Density. After the split, we observe a statistically significant negative correlation in three components. The positive correlation observed in the other five components could potentially be attributed to low contribution alignment, leading to difficulties in managing TD Density.

Conclusion: Our findings suggest that contribution alignment can be important in controlling TD in software development organisations. Making teams responsible for the quality of components they have more expertise over and minimising dependencies between teams can help organisations mitigate the growth of TD.

Keywords: *Ownership and contribution, Contribution degree, Technical debt, Case study*

8.1 Introduction

Large-scale software organisations tend to organise their systems as a constellation of components that are usually developed and maintained by different teams as a way of "componentising" their software architectures. Microservices architecture style [75, 94] is a widely adopted specific example. In a microservices architecture, applications are composed of many small, independent services that communicate with each other, and that are owned, developed, and maintained independently by different teams. This approach can help organisations to enhance maintainability, improve agility, and reduce time-to-market [155]. At the same time, introduce challenges, such as the need for communication and coordination between development teams, and to manage dependencies among microservices and the teams developing them [155].

Although there are several approaches to handling ownership, with different effects on teams' and individuals' autonomy, large-scale organisations usually rely on weak ownership principle [73], which consists of one team *owning* a component (or a microservice). The *owning* team is responsible for the quality of the *owned* component, and its members usually decide about code changes made by members of other teams through code reviews [73]. However, this responsibility does not necessarily mean they are the sole contributors to the code of that component, not even the *main* contributors.

Prior studies explored the relationship between the number of developers in open-source projects and their quality [144, 187]. However, having multiple authors who belong to multiple clusters might have a negative impact on quality, for example, the introduction of security flaws [144]. The quality decline associated with authors from different clusters can be attributed to the concept of *responsibility diffusion*. Responsibility diffusion [18] is a concept from social sciences that describes the lack of action when many people witness a criminal action. All witnesses tend to think someone else will take action (i.e., call the police), and yet, no one does. In software engineering, we associate the concept of responsibility diffusion with the lack of sense of responsibility when a given component (or code element) has multiple authors belonging to different organisations or clusters, and therefore no one will take corrective actions [208]. This article goes one step forward, claiming that this lack of sense of responsibility is also affected by the fact that an owning team does not have a major contribution to the owned code element or component, i.e., the ownership and contribution alignment degree is low.

In proprietary software systems, the alignment between the architecture and the organisation's communication structure becomes a critical factor to success

in the development of large, very large, and ultra-large-scale[2] software systems [24, 155]. Conway [53] hypothesises that *"organisations that design systems tend to produce designs that mirror the communication structure of these organizations"*. It seems natural that when we componentise the architecture, the team constellation and the ownership should be adapted to minimise communication overhead and the dependencies between teams at the task level (i.e., a team depending on some other team's work to finish a task). Ideally, a team will be more productive if it is responsible for the components for which they are the main contributors. This aligns the team's responsibilities with their contribution, which might help reduce the average time to review and integrate new code [203], and help mitigating the accumulation of technical debt (TD). TD is a metaphor used to describe the long-term costs of maintaining and updating a software system due to using suboptimal or inefficient solutions during development [14, 55].

We hypothesise that a low contribution alignment degree can trigger faster accumulation of TD. Teams with low contribution to a given component might not perceive it as their responsibility and might not have enough knowledge to take corrective actions or decide on changes implemented by others.

To study this phenomenon, i.e., the impact that the degree of alignment between ownership and contribution might have on accumulation of TD, we have conducted a case study on a software company developing a very large software system (1.5 million LOC, developed by >20 teams). We have followed the evolution of ten software components (micro-services) initially developed by one team over three years. During this period, the team was split into two, and the components owned by the team were distributed between the new teams. The degree of contribution alignment after the split changed substantially for some components since some of the main contributors were not in the owning team anymore.

The first goal of this case study is to investigate how sudden changes in team constellation might affect teams' effectiveness in managing TD in the components they are responsible for. Second, to analyse the impact of contribution alignment on accumulation of TD.

The remainder of the article is structured as follows: Section 8.2 describes the research methodology, by describing the case, the data collection, and the analysis methods. Section 8.3 presents the results. Section 8.4 discusses the main findings and implications. Section 8.5 discusses and addresses the limita-

[2]Based on the definitions by [62] and [160], respectively.

tions and threats to validity. Section 8.6 discussed the related work. Finally, Section 8.7 presents the conclusion and future work.

8.2 Research Methodology

In this article, we are addressing the following research questions:

- **RQ1:** *How does the change in team structure impact the accumulation of code technical debt?*
- **RQ2:** *How does degree of ownership and contribution alignment impact the accumulation of code technical debt?*

To address the research questions, we have designed an embedded case study (type 2) according to the definition by Yin [225]. We conduct this study in an industrial setting and rely on the analysis of ten `Java` components developed by a company that has chosen to remain anonymous. We use archival data collected from the tools that the company uses during development. The research approach of this article is illustrated in Figure 8.1.

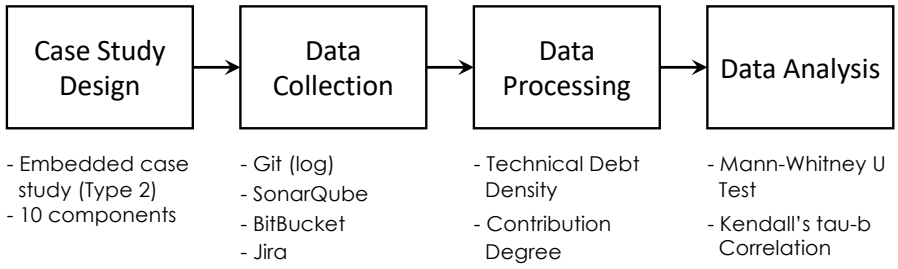

Figure 8.1: The research method.

8.2.1 Case Study Design

The *context* of the case study is a large-scale software development company that develops cloud-based financial and accountancy services. It is a mature company in its development practices and has well-established, successful products on its portfolio, and it has been selected by convenience (availability and access). The company is interested in continuously improving its products and ways of

working. Therefore, it was willing to participate in the study and learn from its results. The *case* is a software system developed by the company with 1.5 million LOC developed by >20 teams. *Units of analysis* are the ten components, i.e., microservices, that are part of this software system. Table 8.1 presents the details of the investigated components in this study.

Table 8.1: Component information - size, number of commits, number of active development weeks, and the owning team.

Component	Size (LOC)	Commits	Active Weeks	Team
C1	18,244	1,019	122	Brown
C2	13,968	714	89	Gray
C3	12,290	872	105	Gray
C4	5,019	192	50	Brown
C5	11,001	691	70	Gray
C6	7,642	366	74	Brown
C7	31,708	1,136	130	Brown
C8	1,872	203	57	Gray
C9	11,994	434	78	Brown
C10	17,187	617	101	Brown
Total:	**130,925**	**6244**	-	-

The analysis period is three years from 2020 to 2022. The measurements are collected in weekly intervals, and we only consider data from the weeks where there was active development on the component. Choosing shorter intervals, e.g., days, or even individual commits, results in a high proportion of observations with `zero` values which might impact the variables under study.

In the following, we describe the main constructs and measurements used for the quantitative analysis of the data collected in this study. We use *Degree of Ownership and Contribution Alignment*, and *Technical Debt Density* ($TD_{Density}$) [1, 59] as the main constructs. Figure 8.2 illustrates the constructs used in this work.

- The *Degree of Ownership and Contribution Alignment* or *Contribution Degree* represents how much of the contribution to a component comes from the officially assigned owner. A 100% contribution degree means that all the contribution to a component is from the formally responsible team (owning team), as designated by the organisation. From this point on, we will refer to the *degree of ownership and contribution alignment*

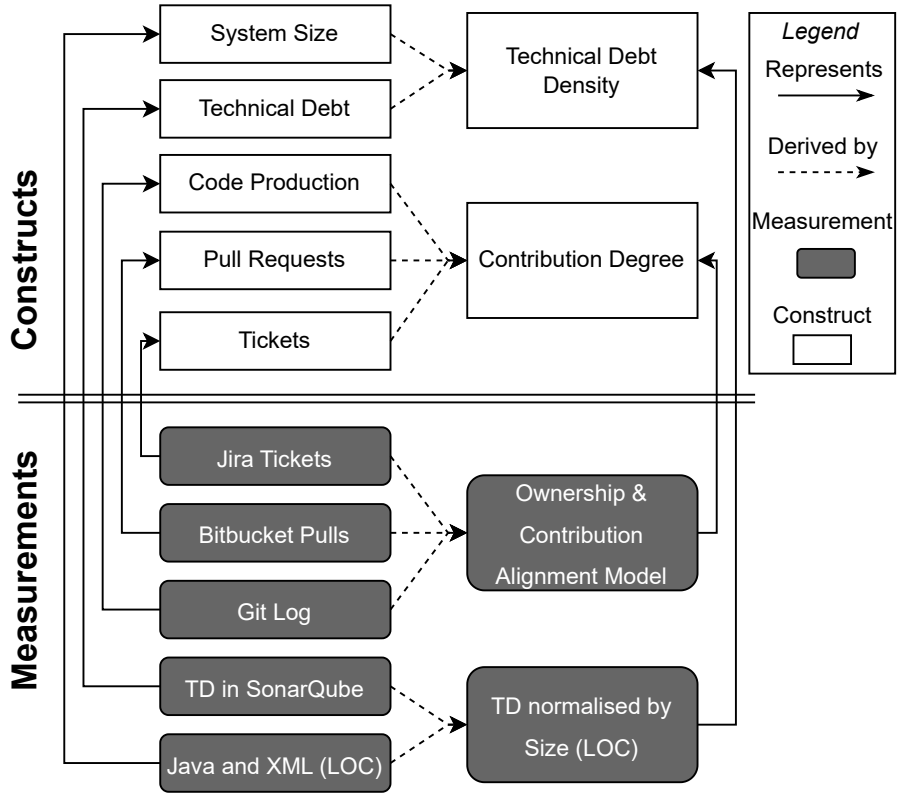

Figure 8.2: Study constructs and measurements.

as *Contribution Degree*. We use the ownership and contribution alignment model (OCAM) [231] to calculate the *Contribution Degree*. In this article, we are using an extension of the original OCAM model reported in [231]. We use four measures out of the seven presented in the original model, but we compile them in a single metric with a ratio scale that allows us to use it as contribution degree, i.e., the ratio of contribution to a component by the owning team. The contribution degree is calculated using the contribution to `code production`, `pull requests`, and `tickets`. 'Code production' is the degree of contribution of a team both

in terms of the number of commits and code churn [150]. 'Pull requests' is the degree of contribution of a team in terms of the number of created pull requests [203]. 'Tickets' is the degree of contribution of a team in terms of the number of tickets created on the ticket management system. The results obtained from applying the OCAM model to the components were validated with component owners and their managers.

- $TD_{Density}$ is the normalised amount of technical debt per line of code [1, 59]. $TD_{Density}$ is calculated by dividing the total amount of TD by the component size (LOC).

The two main variables used in this article are: *Contribution Degree* (See Equation 8.1) and $TD_{Density}$ (See Equation 8.6) $ContributionDegree(t)$ is defined as the contribution degree during a period:

$$Contribution.Degree(t) = \frac{C(t) + Ch(t) + P(t) + T(t)}{4} \tag{8.1}$$

With $C(t)$ being the contribution to the commits (See Equation 8.2), i.e., the percentage of commits done by the team on a specific component; $Ch(t)$ being the contribution to code churn (See Equation 8.3), i.e., the percentages of code written by the team in the code base of a specific component; $P(t)$ being the contribution to pull requests (See Equation 8.4), i.e., the percentage of pull requests created by the team on a specific component; and $T(t)$ being the contribution to tickets (See Equation 8.5), i.e., the percentage of tickets created by the team on a specific component.

$$C(t) = Commit_{Cont.Deg.}(t) = \frac{Commits_{team}}{Commits_{all}} \times 100 \tag{8.2}$$

$$Ch(t) = CodeChurn_{Cont.Deg.}(t) = \frac{CodeChurn_{team}}{CodeChurn_{all}} \times 100 \tag{8.3}$$

$$P(t) = PullRequest_{Cont.Deg.}(t) = \frac{PullRequests_{team}}{PullRequests_{all}} \times 100 \tag{8.4}$$

$$T(t) = Ticket_{Cont.Deg.}(t) = \frac{Tickets_{team}}{Tickets_{all}} \times 100 \tag{8.5}$$

$TD_{Density}(t)$ is defined as the amount of TD t normalised by the component size at a given point in time. It is operationalised as the total remediation time

for all TD items present in the component on a given instant t, as reported by SonarQube, divided by the component size at t:

$$TD_{Density}(t) = \frac{\sum_{i=1}^{\#SystemTDI(t)}(RemeditionTime_i)}{Size(t)} \tag{8.6}$$

8.2.2 Data Collection and Processing

We used widely used tools such as `Git` and API endpoints to access the tools used by the company in their daily operations. We collected data from `Git`[3], `BitBucket`[4], `Jira`[5], and `SonarQube`[6]. We collected the developer team affiliation data from the company's API endpoint. Here, we present how the data was collected from each tool.

- `Git:` We used `Git Log` to collect the data regarding the component size and code production to calculate commit frequency and code churn. Component size is calculated using CLOC[7]. The historical data was collected by checking out the commits and calculating the size using CLOC. When calculating the size of the system, we include only the `Java` and `XML` files. We exclude the other files since technical debt is calculated based on `Java` and `XML` profiles.
- `BitBucket:` We used the BitBucket API to collect the data regarding the creation of pull requests. The raw data was collected using the company API endpoint.
- `Jira:` We used the Jira API to collect the data regarding the creation of tickets. The raw data was collected using the company API endpoint.
- `SonarQube:` We used SonarQube to calculate the amount of code technical debt for each component. The data gathered from SonarQube are the issues detected by the quality profile and quality gate configured by the company. The data is collected via the company's SonarQube instance API endpoint. SonarQube has been used in similar studies on the topic of TD, e.g., [59, 60, 61, 120, 122, 188, 229, 230]. The tool provides the estimated remediation time (i.e., effort in minutes) required to resolve TD

[3]https://git-scm.com
[4]https://bitbucket.org
[5]https://www.atlassian.com/software/jira
[6]https://www.sonarsource.com/products/sonarqube/
[7]https://cloc.sourceforge.net

items (issues in the SonarQube terminology). The accumulated TD of a component at a point in time is the total remediation time for all the issues detected in its codebase. We considered TD as *repaid* when issues are tagged as 'fixed' or 'closed' or are removed from the component[8].

8.2.3 Data Analysis

We used the Shapiro-Wilk test as the normality test to analyse whether *contribution degree* and $TD_{Density}$ 'before' and 'after' the team split were normally distributed[9]. We decided to proceed with non-parametric tests as the data was normally distributed only in one case.

Mann-Whitney U Test for RQ1

To test if the change in team structure impacts the *contribution degree*, we used the Mann-Whitney U test [117, 132] to test to determine whether the contribution degree is different before and after the team split. In other words, we wanted to test whether the team split is a confounding factor that impacts the contribution degree. If the differences between the two periods are statistically significant, the analysis of the relationship between *contribution degree* and $TD_{Density}$ needs to be done separately.

Variables: The independent variable is the categorical groups of 'before' and 'after' the split for a given component. The dependent variable is the Contribution Degree for a given component *in that particular week*.

> *Hypothesis* H_0: There is no difference in contribution degree between 'before' and 'after' teams split.
>
> *Hypothesis* H_1: There is a difference in contribution degree between 'before' and 'after' teams split.

To test if the change in team structure impacts the accumulation of $TD_{Density}$, we used the Mann-Whitney U test. We used the test to examine whether the $TD_{Density}$ is different between 'before' and 'after' the team split.

[8]The company's SonarQube instance has been configured not to remove TDI records when they are removed without tagging them as fixed or closed, but to keep their introduction and removal dates to improve the reliability of the data.

[9]All the statistical tests reported in this subsection were conducted using `Python ver. 3.9.9` and `NumPy ver. 1.10.1`

Variables: The independent variable is the categorical groups of 'before' and 'after' the split for a give component. The dependent variable is $TD_{Density}$ for a given component *in that particular week*.

> *Hypothesis* H_0: There is no difference of $TD_{Density}$ between 'before' and 'after' teams split.
>
> *Hypothesis* H_1: There is a difference of $TD_{Density}$ between 'before' and 'after' teams split.

Kendall's tau-b Correlation Test for RQ2

To test whether there is an association between *Contribution Degree* and $TD_{Density}$ and the type of association, we used Kendall's tau-b correlation coefficient[10]. Kendall's tau-b correlation coefficient is a non-parametric measure of the strength and direction of association between *Contribution Degree* and $TD_{Density}$. Kendall's tau-b correlation coefficient can range from one (1) to minus one (-1). The closer a result is to one, the higher the level of association. The corresponding p-values for Kendall's tau-b correlation coefficients indicate the presence of a significant relationship between *Contribution Degree* and $TD_{Density}$. Kendall's tau-b correlation coefficient is more robust and efficient than alternative non-parametric tests (i.e., Spearman rank correlation), and it is preferred when we have smaller samples.

Variables: Contribution degree for a give component *in that particular week* and $TD_{Density}$ for a given component *in that particular week*.

> *Hypothesis* H_0: There is no association between contribution degree and $TD_{Density}$.
>
> *Hypothesis* H_1: There is an association between contribution degree and $TD_{Density}$.

8.2.4 Validation of the Results

During the execution of the case study, the research team meets the teams under study in recurring meetings every two weeks to report the plan and discuss this and other studies, as well as in focus groups in which we presented the results and collected their feedback. The results of the *contribution degree* were presented to the development team, the engineer leader and product owners to validate the

[10]The correlation tests were conducted using `IBM SPSS Statistics ver. 28`

accuracy of our calculations, and whether they reflected the events that occurred during the studied period (e.g., top contributors leaving the organisation).

Moreover, we gathered data regarding potential explanations for some sudden changes in $TD_{Density}$ during the studied period. In this latter case, to avoid introducing bias, we did not reveal the purpose of the analysis until presenting the full results. This way, we avoid the studied teams trying to defend suboptimal decisions. For this data gathering sessions, we sometimes visualised both non-normalised TD and $TD_{Density}$ to reason about its evolution.

8.3 Results

This section presents the results investigating the impact of ownership and contribution alignment on code technical debt for ten components. Table 8.2 presents the descriptive statistics for *Contribution Degree* and $TD_{Density}$.

As described in Section 8.1, the owning team of the components went through a drastic change and split into two teams on week 9 in 2021 (week 61). Each new team took ownership of certain components. Team Brown took ownership of components C1, C4, C6, C7, C9, and C10, whilst Team Gray took ownership of components C2, C3, C5, and C8 (see Table 8.1). There were other changes to teams' composition during the analysed period, i.e., members leaving a team or new members joining a team. The changes to the teams are illustrated in Figure 8.3.

Figures 8.4 and 8.5 present the box plots for the distribution of *Contribution Degree* and $TD_{Density}$ respectively. We observe:
Contribution Degree: The medians of distributions are different for components *C2*, *C3*, *C4*, and *C10*.
$TD_{Density}$: The medians of distributions are different for components *C1*, *C2*, *C3*, *C6*, *C7*, *C9* and *C10*.

Visual inspection of the components' changes to *Contribution Degree* and $TD_{Density}$ helps identify key events during the analysed period. These key events include: the change in team structure, i.e., the team splitting into two teams on week 9 in 2021 (week 61) and key members leaving a team. For example, a senior architect left Team Gray on week 37 in 2022 (week 141). Figure 8.6 illustrates the evolution of *Contribution Degree* and $TD_{Density}$ for each component. The charts present the changes 'before' and 'after' the team split with blue and orange colours, respectively. The dashed lines illustrate the *Contribution Degree*, and the solid line illustrates $TD_{Density}$.

Table 8.2: Descriptive statistics for contribution degree and $TD_{Density}$. The table is split to present the data separately for before and after the team structure change.

Contribution Degree

	Before					After				
	N	Mean	STD	Min	Max	N	Mean	STD	Min	Max
C1	46	53.183	9.372	31.3	58.9	76	53.271	12.969	36.4	75.9
C2	32	68.516	10.502	52.1	82.3	57	41.853	8.535	32.8	82.4
C3	38	84.482	9.462	68.2	97.2	67	35.278	18.069	20.8	81.7
C4	10	38.240	2.370	36.0	42.5	40	19.230	12.550	2.8	32.8
C5†	-	-	-	-	-	70	83.651	3.672	73.9	89.3
C6	22	35.259	3.879	22.6	39.4	52	37.596	19.230	11.1	55.8
C7	45	42.258	5.502	29.2	45.3	85	40.935	15.336	18.0	62.9
C8†	3	-	-	-	-	54	31.924	32.504	7.3	81.4
C9	26	58.127	7.181	42.4	65.7	52	51.398	7.071	39.5	59.7
C10	36	64.533	2.632	57.9	66.9	65	47.646	8.094	28.8	55.8

Technical Debt Density

	Before					After				
	N	Mean	STD	Min	Max	N	Mean	STD	Min	Max
C1	46	0.221	0.006	0.211	0.231	76	0.205	0.004	0.193	0.213
C2	32	0.232	0.051	0.167	0.392	57	0.158	0.028	0.117	0.203
C3	38	0.093	0.011	0.080	0.126	67	0.099	0.014	0.079	0.112
C4	10	0.085	0.004	0.079	0.089	40	0.085	0.008	0.072	0.094
C5†	-	-	-	-	-	70	0.025	0.036	0.001	0.101
C6	22	0.109	0.007	0.100	0.123	52	0.138	0.006	0.128	0.154
C7	45	0.434	0.007	0.423	0.446	85	0.359	0.034	0.308	0.423
C8†	3	-	-	-	-	54	0.327	0.029	0.278	0.386
C9	26	0.084	0.009	0.074	0.108	52	0.095	0.013	0.073	0.119
C10	36	0.228	0.006	0.215	0.241	65	0.236	0.007	0.223	0.248

†No descriptive statistics due to lack or limited number of observations before team split.

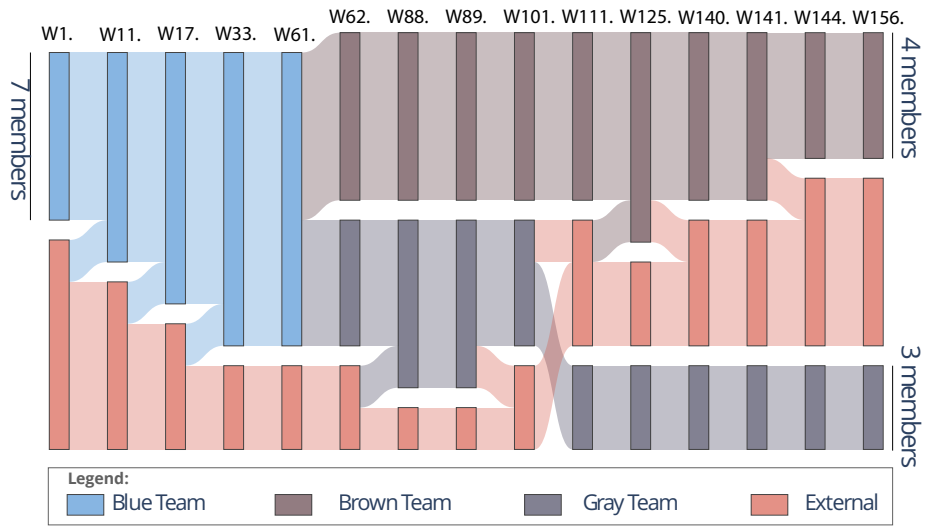

Figure 8.3: Changes to the team during 2020 and 2022. The blue team split into two teams, team brown and team grey, on week 9 in 2021 (week 61).

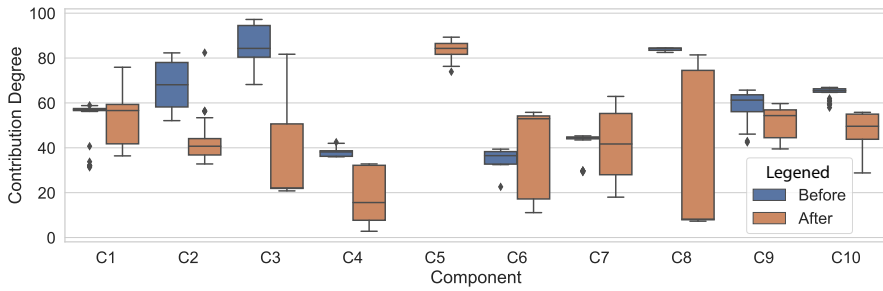

Figure 8.4: Distribution of the *Contribution Degree* observations for components 'before' and 'after' the split.

C1 [`Team Brown`]: The contribution degree for this component slightly increased before the team split. The split caused a sudden drop in contribution since some key developers for that component were allocated to Team Gray. Some weeks after that, the contribution degree went back to levels prior to the split and

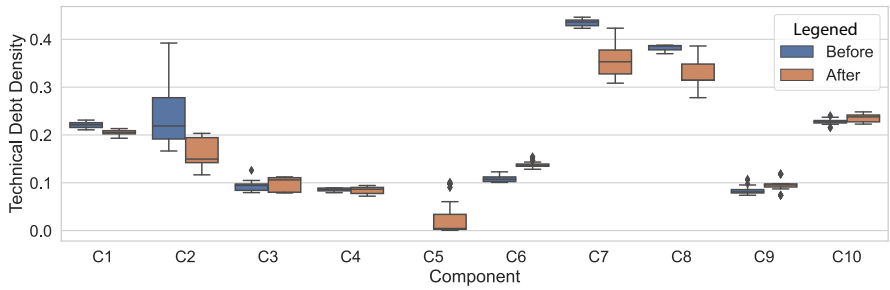

Figure 8.5: Distribution of the $TD_{Density}$ observations for components 'before' and 'after' the split.

continued slightly growing. Conversely, $TD_{Density}$ slightly decreased before and after the team split.

C2 [`Team Gray`]: We observe that $TD_{Density}$ decreases as the contribution degree increases before and after the team split (the team keeping $TD_{Density}$ under control). As in *C1*, there is a major and sudden decrease in contribution degree after the team split. $TD_{Density}$ slightly increases after the split.

C3 [`Team Gray`]: We observe that $TD_{Density}$ slightly decreases as the contribution degree is high before the team split. As in most components, there is a major and sudden decrease in contribution degree when the team splits. And another major and sudden decrease in contribution degree was in week 101 when a product owner[11] left the team. We observe an increasing trend of $TD_{Density}$ during the period after the split.

C4 [`Team Brown`]: We observe that the contribution degree for this component decreases significantly after the team split (close to 0%). $TD_{Density}$ slightly decreases before the team split and increases significantly after the split.

C5 [`Team Gray`]: This component was created after the team split. Therefore, we cannot compare it similar to the other cases. We can observe that the contribution degree is very high from the owning team (80% and higher) from the beginning, meaning that the owning team is its main contributor. As observed in Figure 8.6, $TD_{Density}$ is very low at the beginning of the inception of the

[11]This product owner was a senior developer for over 10 years working on the same components in the same team.

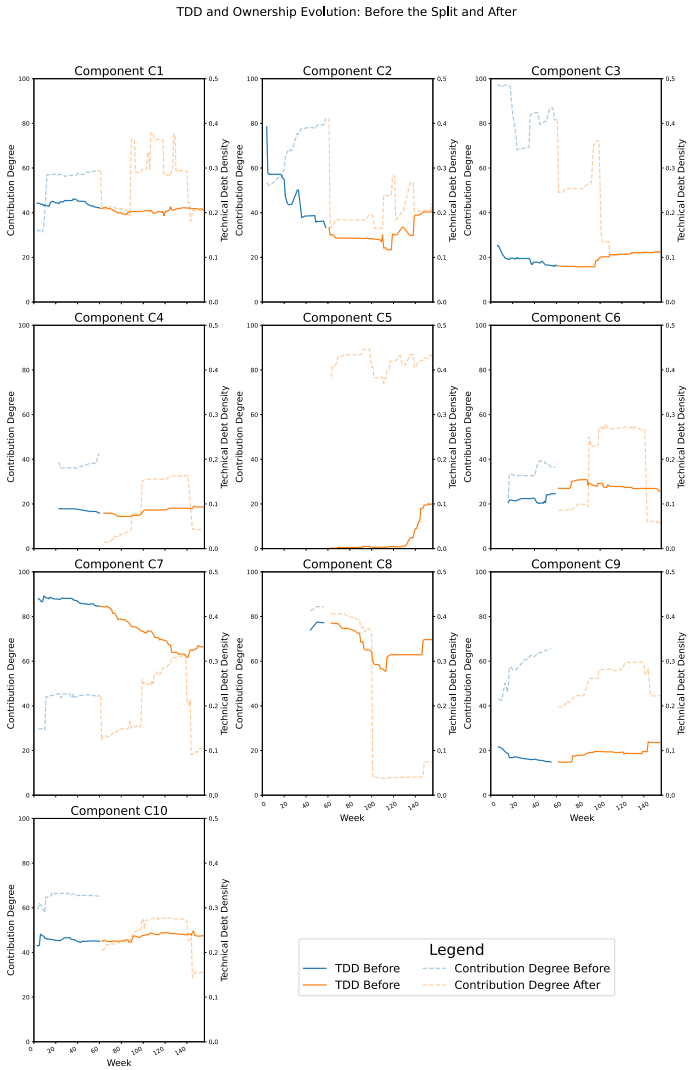

Figure 8.6: The evolution of contribution degree and $TD_{Density}$ in ten components.

component. But it increases towards the end of the analysis, although the ownership levels remain in the same range.

C6 [`Team Brown`]: We observe that the contribution degree for this component decreases after the team split. And $TD_{Density}$ starts to increase after the contribution degree drops. The owning team increases the contribution degree around week 80. This coincides with $TD_{Density}$ slowly decreasing. There is a major and sudden decrease in contribution degree in week 141 when a senior architect left the team.

C7 [`Team Brown`]: This component has the highest amount of $TD_{Density}$ among the investigated components. The component is a fork of an existing open-source repository that the team developed further internally. We observe that $TD_{Density}$ decreases as the owning team increases its contribution degree. There is a major and sudden decrease in contribution degree in week 141 when a senior architect left the team.

C8 [`Team Gray`]: This component was created right before the team split. Contribution degree and $TD_{Density}$ stay do not change significantly before and after the team split. There is a major and sudden decrease in contribution degree in week 101 when a product owner left the team. $TD_{Density}$ started increasing after the product owner left the team.

C9 [`Team Brown`]: We observe that $TD_{Density}$ decreases as the contribution degree increases before the team split. There is a sudden decrease in contribution degree after the team split. The owning team increased their contribution after the team split. $TD_{Density}$ slightly increases after the split.

C10 [`Team Brown`]: We observe that $TD_{Density}$ has not changed substantially after the change. There is a decrease in contribution degree after the team split. However, the owning team has increased their contribution to the component after the split. There is a major and sudden decrease in contribution degree in week 141 when a senior architect left the team.

In order to test whether the change in the team structure significantly impacted contribution degree and $TD_{Density}$, we used the Mann-Whitney U test, as described in Section 8.2.3. The test results show that there is a significant difference between *before* and *after* the team split in contribution degree and $TD_{Density}$ in most of the components (5 out of 8 for contribution degree and 7 out of 8 for $TD_{Density}$). Given that the team split is a confounding factor for analysing the association between contribution degree and $TD_{Density}$, we decided to separately perform Kendall's tau-b correlation test for the periods

before and after the team split. The test results are presented in Table 8.3. The statistically significant cases are highlighted in bold.

Table 8.3: Mann-Whitney U test results for contribution degree and technical debt density to compare the differences between *before* and *after* the teams split. Significant results are presented in boldface.

	Contribution Degree		**Technical Debt Density**	
	P-value	N	P-value	N
C1	0.571	122	<**0.001**	122
C2	<**0.001**	89	<**0.001**	89
C3	<**0.001**	105	**0.029**	105
C4	<**0.001**	50	0585	50
C5†	-	-	-	-
C6	0.195	74	<**0.001**	74
C7	0.922	130	<**0.001**	130
C8†	-	-	-	-
C9	<**0.001**	78	**0.001**	78
C10	<**0.001**	101	<**0.001**	101

†No statistical test due to lack or limited number of observations before team split.

The results from Kendall's tau-b correlation coefficient for components 1-10 are presented in Table 8.4. The results depict if there are any differences under two different occasions: before and after the team split. The statistically significant cases are highlighted in bold. The p-values less than 0.01 are identified with (§), and p-values that are less than 0.05 are identified with (†).

The results for the correlation (see Table 8.4), before the team split up, show that there is a statistically significant relationship between *contribution degree* and $TD_{Density}$ for components *C1*, *C2*, *C4*, and *C9*. For component *C5*, we have no observations before the team split. We have very few observations for component *C8*, i.e., 3 cases. Thus, we present no results for those two components before the team split.

On the other hand, in the results (see Table 8.4) for the correlation after the team split, a statistically significant relationship between contribution degree and $TD_{Density}$ was detected for all the components except for component *C5*, where the team did not split, and component *C9*.

Figure 8.7 illustrates the relationship between *contribution degree* (X axis) and $TD_{Density}$ (Y axis) for each component. We use Kendall's tau-b correlation

Table 8.4: Test results - Kendall's τ. Significant results are in boldface. Magnitude of association is calculated based on [36].

	Before				**After**			
	P-value	Kendall's τ	Magnitude	N	P-value	Kendall's τ	Magnitude	N
C1	**0.001**	**-0.332** §	**Strong**	**46**	**0.013**	**-0.199** †	**Moderate**	**76**
C2	**<0.001**	**-0.840** §	**Strong**	**32**	**0.011**	**0.237** †	**Moderate**	**57**
C3	0.410	0.093	Very Weak	38	**<0.001**	**-0.353** §	**Strong**	**67**
C4	**0.037**	**-0.529** †	**Strong**	**10**	**0.002**	**0.353** §	**Strong**	**40**
C5*	-	-	-	-	0.073	0.148	Weak	70
C6	0.051	-0.301	Strong	22	**0.019**	**0.229** †	**Moderate**	**52**
C7	0.309	-0.107	Weak	45	**<0.001**	**-0.464** §	**Strong**	**85**
C8*	-	-	-	3*	**<0.001**	**0.770** §	**Strong**	**54**
C9	**0.001**	**-0.918** §	**Strong**	**26**	0.282	0.104	Weak	52
C10	0.743	0.039	Very Weak	36	**<0.001**	**0.454** §	**Strong**	**65**

†Correlation is significant at the 0.05 level (2-tailed).
§Correlation is significant at the 0.01 level (2-tailed).
* No statistical test due to lack or limited number of observations before team split.

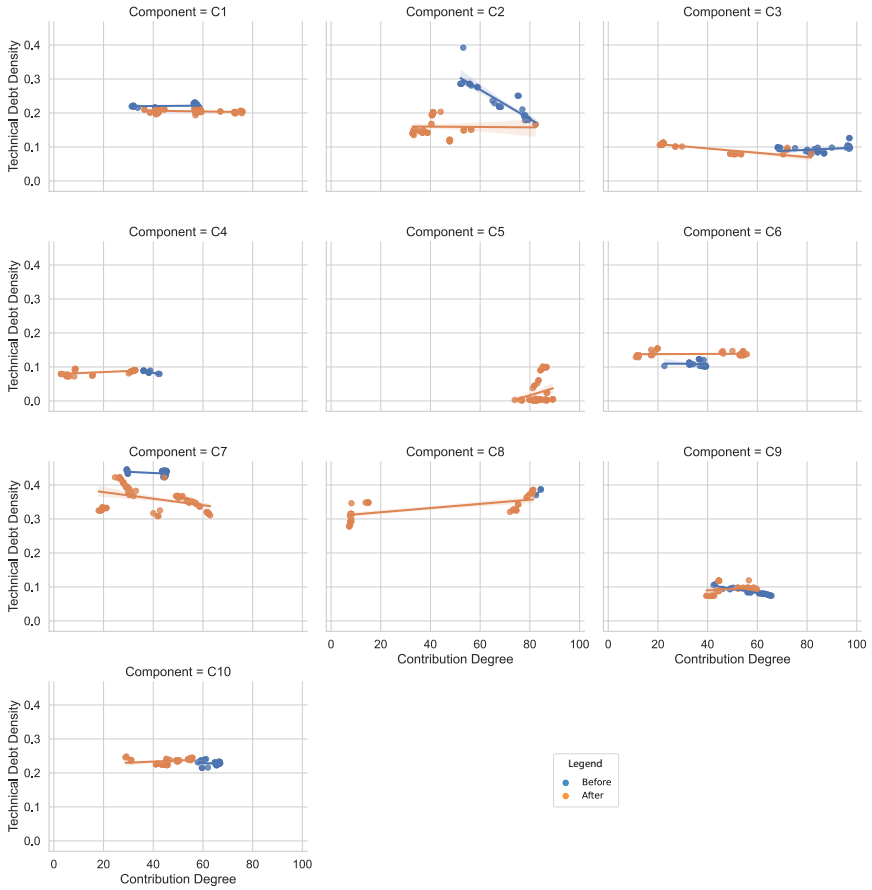

Figure 8.7: Relationship between *Contribution Degree* (X axis) and $TD_{Density}$ (Y axis) for each component. The regression lines presented in the figure are only for visualisation. We are not using regression to describe the results nor prediction of contribution degree and $TD_{Density}$.

coefficient (see Table 8.4) and Figure 8.7 to investigate each component. Note that the regression lines presented in the figure are used only for visual inspection. We are not using regression to draw conclusions nor predict contribution degree and $TD_{Density}$.

C1 [`Team Brown`]: There is a significant strong negative correlation between contribution degree and $TD_{Density}$ before the team split (P-Value: 0.001; Kendall's τ: -0.332) and a significant moderate negative correlation after team split (P-Value: 0.013; Kendall's τ: -0.199). Our observation and the test results suggest that as the contribution degree increases over time, $TD_{Density}$ decreases in the component.

C2 [`Team Gray`]: There is a significant strong negative correlation between contribution degree and $TD_{Density}$ before the team split (P-Value: < 0.001; Kendall's τ: -0.840) and a significant moderate positive correlation after team split (P-Value: 0.011; Kendall's τ: 0.237). We observe a different pattern after the team split. There is a significant moderate positive correlation between contribution degree and $TD_{Density}$; as contribution degree increases, $TD_{Density}$ increases too.

This increase in $TD_{Density}$ might be due to the context of this particular component. The component is related to handling *user authentication* in the system. There were issues integrating a third-party authentication application into the system and the increase in $TD_{Density}$ can be attributed to the development of new features.

Furthermore, we observe that as the contribution degree decreases to 40% and below, $TD_{Density}$ changes unexpectedly, i.e., other factors such as responsibility diffusion [208] and knowledge loss [35, 205] might have a bigger impact on $TD_{Density}$.

C3 [`Team Gray`]: There is no significant correlation between contribution degree and $TD_{Density}$ before the team split (P-Value: 0.410; Kendall's τ: 0.093). However, the results show a significant strong negative correlation after the team split (P-Value: < 0.001; Kendall's τ: -0.353). Our observation and the test results suggest a decrease in $TD_{Density}$ as the contribution degree increases only before the split.

Similar to *C2*, there is a decrease of contribution degree below 40% around week 100 in this component, after which $TD_{Density}$ starts increasing.

C4 [`Team Brown`]: There is a significant strong negative correlation between $TD_{Density}$ and contribution degree before team split (P-Value: 0.037; Kendall's τ: -0.529) and a significant strong positive correlation after team split (P-Value: 0.002; Kendall's τ: 0.353). Overall, the team's contribution degree is low in this component.

We observe a different pattern after the team split with a significant strong positive correlation between contribution degree and $TD_{Density}$. This increase in $TD_{Density}$ might be due to the context of this component, which is related to

handling different *licenses* in the system. There has been a significant amount of new development in this component, and the increase in $TD_{Density}$ can be attributed to the development of new features while the owning team was gaining ownership.

We observe a decrease of contribution degree to near 0% after the team split, after which $TD_{Density}$ starts increasing.

C5 [`Team Gray`]: There is no significant correlation between $TD_{Density}$ and contribution degree since the creation of this component (P-Value: 0.073; Kendall's τ: 0.148).

C6 [`Team Brown`]: There is no significant correlation between $TD_{Density}$ and contribution degree before the team split (P-Value: 0.051; Kendall's τ: -0.301). However, the results show a significant moderate positive correlation after the team split (P-Value: 0.019; Kendall's τ: 0.229). There are two clusters for contribution degree observations —orange dots— after the team split (See Figure 8.7 Component C6). This left cluster, i.e., lower contribution degree, is related to the fact that a senior architect left the team. The increase of the $TD_{Density}$ after the team split is also associated with the same event.

C7 [`Team Brown`]: There is no significant correlation between $TD_{Density}$ and contribution degree before the team split (P-Value: 0.309; Kendall's τ: -0.107). However, there is a significant strong negative correlation after the team split (P-Value: < 0.001; Kendall's τ: -0.464). Our observation and the test results suggest that as the degree of ownership increases over time after the split, $TD_{Density}$ decreases in the component.

Similar to *C2*, *C3*, and *C4*, we observe a decrease of contribution degree to below 40% around week 145 in this component, after which $TD_{Density}$ starts increasing.

C8 [`Team Gray`]: There is a significant strong positive correlation between $TD_{Density}$ and contribution degree after the team split (P-Value: < 0.001; Kendall's τ: 0.770). There are two clusters for contribution degree observations —orange dots— after the team split (See Figure 8.7 Component C8). The left cluster, i.e., lower contribution degree, is related to the removal of a product owner from the team. The increase of the $TD_{Density}$ after the team split is also associated with the same event.

Similar to *C2*, *C3*, *C4* and *C7*, we observe a decrease of contribution degree to below 40% around week 100 in this component, after which $TD_{Density}$ starts increasing.

C9 [`Team Brown`]: There is a significant strong negative correlation between $TD_{Density}$ and contribution degree before the team split (P-Value: 0.001; Kendall's τ: -0.918). However, there is no significant correlation after the team split (P-Value: 0.282; Kendall's τ: 0.104). We observe a decrease of contribution degree to below 40% after the team split, after which $TD_{Density}$ starts increasing.

C10 [`Team Brown`]: There is no significant correlation between $TD_{Density}$ and contribution degree before the team split (P-Value: 0.743; Kendall's τ: 0.039). However, the results show a significant strong positive correlation after the team split (P-Value: < 0.001; Kendall's τ: 0.454). There is a significant strong positive correlation between contribution degree and $TD_{Density}$. This increase in $TD_{Density}$ is due to the context of this component. The component is related to handling *users* in the system. The team maintains three versions of the same functionality at the same time, and the increase in $TD_{Density}$ can be attributed to it.

8.4 Discussion

This section discusses the insights gained from analysing the components and practical implications of the findings.

> **RQ1.** *How does the change in team structure impact the accumulation of code technical debt?*

The findings of this study suggest that altering the structure of a team, specifically by dividing it into two separate teams, has the potential to impact the accumulation of technical debt. Specifically, our analysis reveals that components *C2*, *C4*, *C6*, and *C10* experienced a reduction in their contribution degree to below 40%, which corresponded with an increase in $TD_{Density}$ in the subsequent weeks. Our observations are similar to the work presented by Nagappan et al. [153] on the impact of organisation structure on code quality.

The observations were further corroborated by the statistical tests conducted. However, it is worth noting that we cannot draw the same conclusion for component *C8*, as its introduction occurred several weeks prior to the team split and thus was not subject to the same impact. These findings highlight the importance of carefully considering the potential effects of team restructuring on technical debt accumulation and the need for continued monitoring and analysis to ensure optimal team performance and productivity.

RQ2. *How does degree of ownership and contribution alignment impact the accumulation of code technical debt?*

The results presented in Section 8.3 suggest that contribution degree might impact the accumulation of $TD_{Density}$. We observe that before the split that in four components (*C1*, *C2*, *C4*, *C9*)[12] the contribution degree has a negative correlation, meaning that the higher the contribution degree, the lower $TD_{Density}$. After the split, we observe eight components for which the contribution degree seems to impact $TD_{Density}$. In components *C1*, *C3*, and *C7*, we observe a statistically significant negative correlation between contribution degree and $TD_{Density}$; the higher the contribution degree, the lower $TD_{Density}$. However, we also observe that for components *C2*, *C4*, *C6*, *C8*, and *C10*, there is a positive correlation, which might seem counter-intuitive. The first clarification is that a lower contribution degree does not mean not incurring in TD since many other factors might be impacting its accumulation, but what we observe, after inspecting Figures 8.6 and 8.7, that $TD_{Density}$ tend to grow together with contribution degree when the contribution degree is at levels below 40% (except for the case of *C5*), which might mean that controlling TD effectively is harder when the work in that component is mainly done by developers from outside the team.

One aspect to be considered when analysing the results after the split is that the change in trend on $TD_{Density}$ might manifest with a few weeks of lag. In the first weeks after the split, the owning team might restrain themselves from making big changes, or the changes we observe might have been under development before the split and therefore have a different impact on TD.

In the case of *C5*, There might be several explanations, the first being the fact that during the first weeks of development for a component created from scratch, $TD_{Density}$ is more easily kept under control, and when the functionality starts to grow $TD_{Density}$ grows with it. We can also explain with the *Technical Credit* concept. Experienced teams or individual developers might have *credit* to incur in TD, similar to the financial concept of *credit*, which allows a customer to loan money from a financial institution. Higher ownership might be linked with technical credit, which we will study in further work. C8 does not seem to follow the same pattern because *C8* is created a few weeks before the split,

[12]For components *C5* and C8 where there is no analysis before the split since *C5* was a component created after the team split and *C8* only a few weeks before, and therefore without enough data points to draw meaningful conclusions.

but not from scratch. Instead, this component was created as a migration of a service in a monolith architecture to a microservices.[13]

We observe that when a key contributor (senior architect and senior developer) leaves the team, the impact of "architectural knowledge vaporisation" [35, 205] might be manifested as an increase in technical debt. This result is aligned with the findings by Bird et al. [32], which suggest that high levels of top contributors result in higher quality. We observed that when a senior developer left the team in Week 101, the contribution degree dropped more than 50%, in components *C3* and *C8*, resulting in an increase in $TD_{Density}$. Similarly, we observe the same when a senior architect left the team in Week 141, the contribution degree dropped more than 50% in components *C6* and *C7*.

Our observations suggest that the events that cause major changes to contribution degree, i.e., the decrease of contribution degree, might impact the accumulation of $TD_{Density}$. As reported by [57], software quality is directly linked to the volume of collaboration and commitment of the development team. Therefore, the impact of a major decrease in contribution degree can be manifested as the faster accumulation of technical debt (i.e., $TD_Density$ growth over time).

Implications

- In the process of assigning component ownership to teams, it is crucial to take into account the contribution degree and its impact on the accumulation of TD to each component. The changes in contribution degree can arise from a number of factors, including organisational changes —changes to team constellation— and attrition —team members leaving a team or the company—, among others. Neglecting to consider these factors when making component ownership decisions can lead to increase in TD impacting the development effectiveness and efficiency. Therefore, it is imperative for managers and team leaders to carefully evaluate the contribution degree to each component.

- In the context of organisational restructuring, when the creation of new teams involves splitting existing ones, a key consideration is assigning, when possible, components to the sub-team that exhibits the highest degree of contribution. This rationale may be based on a number of factors,

[13] Most of the components under study are part of a migration from a monolith to a microservices architecture, but they count on a long development history. However, C8 is the only component that results from the migration of functionality from the monolith that occurred during the period under analysis.

such as the distribution of expertise and skills among team members, the complexity of the component, or the urgency of the task at hand. By assigning components in this manner, organisations can aim to minimise the impact of the uncontrolled accumulation of TD.

- The contributions of the senior members of the team are important to preserving the quality of the code [170]. Attrition is unavoidable and members of the team might leave due to a variety of reasons. Companies and development teams should understand the degree of contribution of individual team members to try to preserve knowledge and avoid architectural knowledge vaporisation.
- In general, high contribution degree levels seem to be a good way to manage TD effectively. We hypothesise that a high contribution degree is a way to fight back responsibility diffusion. We might have many other teams contributing to a particular component, but if the owning team is also the main contributor, they might put more effort into caring about the code base and performing TD mitigation activities, e.g., refactorings [199].
- Ownership and contribution alignment and its implications (i.e., TD) seem to be phenomena that need to be studied further, especially in proprietary software development. This might also help us further understand the potential impact of organisational changes.
- We observed that experienced and skilled members of teams, such as senior architects and senior developers, tend to make decisions that may manifest as technical debt. These individuals possess a high degree of expertise and can take on more debt due to their seniority and major contributions. Hence, the term *Technical Credit* can be used to describe their ability, potential, and magnitude of making sub-optimal decisions that may result in TD.

8.5 Limitations and Threats to Validity

Different types of threats to validity might impact the results of our research. *Construct validity* concerns the selection of measurements that reflect the constructs. The study constructs and measurements used in this study are presented in Section 8.2. We use several measurements from different tools including `Git`, `BitBucket`, `Jira`, and `SonarQube`. We are aware of the limitations

of static analysis tools (e.g., SonarQube) in detecting TD items [121], however first, SonarQube is the tool (and construct) that the studied company (and many others) uses for reasoning about technical debt and code and architectural degradation. Second, it has been used in other studies approaching the evolution of Technical Debt [59, 60, 61, 120, 122, 188, 230]. We use $TD_{Density}$ [1] and *contribution degree*. The former has been used in other studies to investigate the impact of external factors such as Clean Code [60] and the latter is an extension of the model presented in [231]. We do not distinguish between the TD items when conducting the analysis, i.e., all TD items are considered to have the same weight in the calculations. The repaid TD is calculated through the issues flagged as 'fixed' and 'closed'. However, there might be cases where TD has been removed by deleting the files that might not be included in our analysis.

We mitigated the potential threats regarding the calculations of measures and the selection of cases in consensus with the heads of development in the studied company.

Internal validity concerns the extent to which the results are free from error, i.e., the results represent the truth. A major threat to the validity of this study is the presence of confounding factors that, together with ownership and contribution misalignment, might impact the results presented in this work. We present the results from correlation tests, i.e., the results presented in this article do not aim to provide a cause-and-effect relationship between the studied variables. We mitigate this threat to validity by providing contextual information regarding the components to find other possible factors impacting our observations. There might be other non-studied factors that can explain the observations. Additional studies are required to better understand the other factors impacting the growth of $TD_{Density}$.

External validity concerns the generalisability of the results. The study results presented in this article only apply to the components investigated in this case study. The case study results should not be considered generalisable based on cases. We use statistical tests strictly to reason about each specific case.

Reliability, as one of the biggest threats to validity in this study, concerns the collected data and the performed analysis to be independent from individual researchers. We mitigated this threat to validity by consistently and frequently involving the developing teams —holding bi-weekly meetings— throughout the study design and interpreting the results.

Finally, *conclusion validity* refers to the degree to which the conclusions drawn from a statistical analysis are accurate and reliable [191]. It is concerned

with whether the statistical inferences are based on a sound and robust data analysis. To mitigate this threat, we used appropriate statistical methods and considered potential confounding variables that could affect the results, i.e., team split. We also collected an adequate sample size representative of the population being studied.

8.6 Related Work

There are several research works that scrutinise the relationship between team or organisation structure and software quality [32, 41, 144, 153, 170, 172, 187, 203], some of which directly put the focus on the relationship of ownership and software quality, e.g., [32, 170, 172, 203]. The concept of $TD_{Density}$ [1, 13, 26, 59, 125] has been used as a construct to reason about the evolution of code and architectural degradation [228]. The degree to which developers contribute to a particular project has also been addressed in several studies, e.g., [57, 83, 161, 168] as a way to characterise the degree in which developers participate in software projects.

In the following subsections, we summarise relevant works in each of these areas to position the contribution of our study in relation to the existing literature on the field.

8.6.1 Organisations, Team Structure and Quality

The quality of any product is strongly affected by organisation structure [41]. This is empirically shown by Nagappan et al. [153] on the relationship between organisational structure and software quality via eight organisational complexity metrics from the code viewpoint. Examples include "*the absolute number of unique engineers who have touched a binary and are still employed by the company*" and "*the total number of unique engineers who have touched a binary and have left the company as of the release date of the software system*" [153]. The authors develop a model to predict the failure-proneness and achieve a precision of up to 86.2% and recall of up to 84%, compared to other traditional metrics, including code churn and code complexity. Thus, organisational and owner metrics are more effective in software quality estimation.

There also have been studies analysing the impact that the number of developers might have on the quality of Open-Source Systems (OSS) (e.g., [144, 187]). Schweik et al. [187] opposes two laws (or principles): Brooks' law [41] against Linus' law [176]. On the one hand, Brooks' law states that adding more (hu-

man) resources to a software project tends to make it late. On the other hand, Linus' law [176] named after the creator of the Linux operating system, states that "with enough eyeballs, all bugs are shallow". In other words, adding more developers (in OSS) encompasses higher quality. In [187], authors report the impact of the number of developers on the "survivability" of OSS projects, using the latter as a proxy of its quality. Meneely and Williams [144] did a similar study, in this case just focusing on Linus' law, specifically regarding the number of vulnerabilities. The main result is that the presence of developers from different clusters seems to have a negative impact on quality in the form of more vulnerabilities present in the system.

8.6.2 Software Ownership and Quality

Bird et al., [32] examine the relationship between various (software) ownership measures (e.g., *the number of low-expertise developers* and *proportion of ownership for the top owner* - although in this case, the authors address individual contributors and not teams) and software failures, and find that the measures of ownership are associated with pre-release faults and post-release failures. In particular, the evidence of correlation is significant for minor contributors, i.e., those who contribute less and infrequently on a module. Our *contribution degree* calculation uses the degree of contribution of the owning team, instead of the top contributor, as the metric to reason about the degree of alignment between ownership and contribution, and its impact on $TD_{Density}$

Thongtanunam et al. [203] argue that code ownership comes with responsibility, and developers who author the majority of changes to a module are presumably the owners. However, contribution may come in other forms, e.g., in modern code review by criticising code changes authored by other developers. In our *contribution degree* calculation, we consider code review information. The authors also show that 67%-86% of the developers who contribute to a module do not author code changes, i.e., contribute by reviewing code. Among them, only 18%-50% are core team members. Thus, the authors suggest that code review should be included in code ownership estimation. While evaluating the relationship between review-specific and review-aware code ownership and defect-proneness, the authors find that developers with low traditional and review ownership are more prone to post-release faults. These results aligned with our findings that suggest that higher ownership might help teams keep $TD_{Density}$[14].

[14]One of the dimensions used by SonarQube, which includes bugs, as detected using static analysis, which is different from the number of bug reports in ticket management systems.

Moreover, Posnett et al. [170] argue that low ownership of a module (i.e., too many contributors) can affect code quality. The authors define a metric called DAF (Developer's Attention Focus, which measures the focus of a developer on some activities) and show that more focused developers introduce fewer defects than less-focused developers. In contrast, files receiving narrowly focused activity are more likely to have faults than others.

8.6.3 Technical Debt Density in Source Code

In an exploratory study, Al Mamun et al. [1] investigates the ability of the '$TD_{Density}$ trend' metric to estimate the evolution of technical debt (TD). The '$TD_{Density}$ trend' is the slope of two consecutive '$TD_{Density}$' measures. Using the $TD_{Density}$ trend' metric, the authors observe that a file has the highest level of $TD_{Density}$ at the initial stage of its revisions, and $TD_{Density}$ decreases as the file size increases.

In another study, Digkas et al.[59] examine the relationship between the amount of technical debt in new code and the evolution of technical debt in a system. The authors argue that TD grows in absolute numbers as software systems evolve. In contrast, the density of TD (i.e., TD divided by lines of code, also known as normalised TD) may decline due to refactoring or the development of new artefacts with less TD. As their findings, the authors report that among the three major types of code changes (insertion, deletion, and modification), the contribution of code modification (i.e., refactoring) is strongly related to the change in the $TD_{Density}$.

Levén et al. [125] investigate the causal relationship between the existing $TD_{Density}$ of a system and developers' tendency to introduce new TD during the system's evolution (using the Broken-Window Theory). The findings suggest that existing TD affects developers' tendency to introduce new TD during further system development.

Arvanitou et al. [13] investigate how the accumulation of TD can be explained in (scientific) software development. To do that, authors first identify software engineering practices used to develop scientific software and the most common causes of introducing TD and then map them. Findings suggest that the scientists that develop scientific software lack certain software engineering practices and introduce TD in scientific software. To minimise the TD in scientific software, the authors recommend reusing software libraries and process improvement methodologies and working in pairs (i.e., applying eXtreme Programming), which would minimise about half of the causes of introducing TD.

8.6.4 Developer Contribution Metrics and Quality

Developers' contributions can be estimated and likewise can be linked to the quality of the source code. De Bassi et al. [57] argue that software quality can be directly linked to the volume of collaboration and commitment in the development team. The authors evaluate 20 quality metrics related to complexity, inheritance, and size that can measure team members' participation regarding their source code contribution to the project. In the end, those metrics are analysed based on positive, negative, and no influence on source code quality, including maintainability, testability, and understandability.

Parizi et al. [168], utilising git-driven technology and its features, estimate and visualise a team member's contribution using a set of metrics, including *the number of commits*, *number of merge pull requests*, *number of files*, and *total lines of code*. For computing these metrics, authors primarily rely on git logs as inputs to extract the performance data in combination with the total time spent on a project each day.

In another work, Oliveira et al. [161] define source code ownership by a developer when they contributed most to a source code file. The authors study several code-based and commit-based developer productivity metrics, including *source lines of code by time* (SLOC/Time, code-based) – the larger the source code created by the developer, the higher their productivity; and *commits performed by time* (Commits/Time, commit-based) – the higher the number of commits, the higher is their productivity. Using those metrics, the authors explore whether and to what extent developer productivity metrics have a relationship with developers' productivity. Authors find that code-based metrics better explain productivity than commit-based metrics.

The literature shows that in software development, team structure, and the developers' contribution seem to impact quality. However, what is not known is whether the misalignment between ownership and contribution (i.e., low values of *contribution degree*) affects TD. This article aims to bridge this gap by studying the relationship between the TD and the misalignment between ownership and contribution. To measure the misalignment between ownership and contribution, we include several perspectives, including contributions to code reviews as suggested by [203], and measure contribution degree based on the degree of contribution by the owning team, as suggested by [32].

8.7 Conclusion

In this article, we present the results of an industrial case study on the impact of ownership and contribution alignment on code technical debt. We investigated ten components during a three-year period (2020 to 2022) from a large software development company that develops web-based financial and accountancy services.

Study results reveal that prior to the team structure's change, a negative correlation was found between contribution degree and $TD_{Density}$ in the majority of cases where high levels of contribution degree were present. This negative correlation means that higher contribution degree correlate with lower $TD_{Density}$ was statistically significant in four components, indicating that heightened contribution degree was associated with decreased $TD_{Density}$. Once the original team was divided into two, a statistically significant negative correlation was observed in three components, whereas five components exhibited a positive correlation that may be attributed to low levels of contribution degree, thereby suggesting the team's inability to effectively manage $TD_{Density}$.

The results suggest that, when assigning the components' ownership responsibility to teams, it is important to consider the contribution degree of each team to make sure that the team receiving this responsibility is ready to take the required actions to preserve the components' $TD_{Density}$ under control. Additionally, the contributions of senior team members might be crucial to mitigating the accumulation of code TD and preserving knowledge when attrition occurs. High levels of contribution degree can help manage TD effectively by restraining responsibility diffusion and encouraging the owning team to invest more effort in mitigating TD. The study highlights the need for further research on the ownership and contribution alignment and their impact on TD in proprietary software development, which may help organisations better understand the potential effects of organisational changes.

However, both concepts, ownership and contribution degree alignment and TD, are complex phenomena, and other factors might impact the changes in the accumulation of TD. In addition, we are aware that the results are not generalisable, but the aim of this paper is also to raise awareness of the importance of aligning ownership and contribution degree.

We believe the results presented in this article need to be strengthened with future replications of the study. Finally, we plan to investigate the concept of TD credit in other cases.

References

[1] Al Mamun, M. A., Martini, A., Staron, M., Berger, C., and Hansson, J. Evolution of technical debt: An exploratory study. In *2019 Joint Conference of the International Workshop on Software Measurement and the International Conference on Software Process and Product Measurement, IWSM-Mensura 2019, Haarlem, The Netherlands, October 7-9, 2019* (2019), vol. 2476, CEUR-WS, pp. 87–102.

[2] Alégroth, E., and Gonzalez-Huerta, J. Towards a Mapping of Software Technical Debt onto Testware. In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications* (Vienna, Austria, 2017), pp. 404–411.

[3] AlOmar, E. A., Rodriguez, P. T., Bowman, J., Wang, T., Adepoju, B., Lopez, K., Newman, C., Ouni, A., and Mkaouer, M. W. How do developers refactor code to improve code reusability? In *Reuse in Emerging Software Engineering Practices: 19th International Conference on Software and Systems Reuse, ICSR 2020, Hammamet, Tunisia, December 2–4, 2020, Proceedings 19* (2020), Springer, pp. 261–276.

[4] 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), 100–121.

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

[6] Amadi-Echendu, J. E., Brown, K., Willett, R., and Mathew, J. *Definitions, Concepts and Scope of Engineering Asset Management*, vol. 1. Springer London, London.

[7] Amanatidis, T., Mittas, N., Moschou, A., Chatzigeorgiou, A., Ampatzoglou, A., and Angelis, L. Evaluating the agreement among technical debt measurement tools: building an empirical benchmark of technical debt liabilities. *Empirical Software Engineering 25* (2020), 4161–4204.

[8] Ampatzoglou, A., Ampatzoglou, A., Avgeriou, P., and Chatzigeorgiou, A. A Financial Approach for Managing Interest in Technical Debtb. In *International Symposium on Business Modeling and Software Design* (2016), pp. 117–133.

[9] 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), 52–73.

[10] Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., Avgeriou, P., and Stamelos, I. Reusability index: A measure for assessing software assets reusability. In *International Conference on Software Reuse* (2018), Springer, pp. 43–58.

[11] Arcoverde, R., Garcia, A., and Figueiredo, E. Understanding the longevity of code smells: preliminary results of an explanatory survey. In *Proceedings of the 4th Workshop on Refactoring Tools* (2011), ACM, pp. 33–36.

[12] Arvanitou, E.-M., Ampatzoglou, A., Bibi, S., Chatzigeorgiou, A., and Stamelos, I. Monitoring technical debt in an industrial setting. In *Proceedings of the Evaluation and Assessment on Software Engineering* (2019), ACM, pp. 123–132.

[13] Arvanitou, E.-M., Nikolaidis, N., Ampatzoglou, A., and Chatzigeorgiou, A. Practitioners' perspective on practices for preventing technical debt accumulation in scientific software development. In *ENASE* (2022), pp. 282–291.

[14] Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. Managing technical debt in software engineering (dagstuhl seminar 16162). In

Dagstuhl Reports (2016), vol. 6, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[15] Avgeriou, P. C., Taibi, D., Ampatzoglou, A., Fontana, F. A., Besker, T., Chatzigeorgiou, A., Lenarduzzi, V., Martini, A., Moschou, A., Pigazzini, I., et al. An overview and comparison of technical debt measurement tools. *Ieee software 38*, 3 (2020), 61–71.

[16] Badampudi, D., Wohlin, C., and Petersen, K. Experiences from using snowballing and database searches in systematic literature studies. In *Proceedings of the 19th International Conference on Evaluation and Assessment in Software Engineering* (2015), pp. 1–10.

[17] Bao, L., Li, T., Xia, X., Zhu, K., Li, H., and Yang, X. How does working from home affect developer productivity?–a case study of baidu during covid-19 pandemic. *arXiv preprint arXiv:2005.13167* (2020).

[18] Barley, J. M., and Latanfi, B. Bystander intervention in emergencies: Diffusion of responsibility. *Journal of Personality and Social Psychology 8* (1968), 377–383.

[19] Bass, L., Clements, P., and Kazman, R. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[20] Bass, L., Clements, P., and Kazman, R. *Software architecture in practice*. Addison-Wesley Professional, 2021.

[21] Bavani, R. Distributed agile, agile testing, and technical debt. *IEEE software 29*, 6 (2012), 28–33.

[22] Bavota, G., De Carluccio, B., De Lucia, A., Di Penta, M., Oliveto, R., and Strollo, O. When does a refactoring induce bugs? an empirical study. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation* (2012), IEEE, pp. 104–113.

[23] Bavota, G., De Lucia, A., Di Penta, M., Oliveto, R., and Palomba, F. An experimental investigation on the innate relationship between quality and refactoring. *Journal of Systems and Software 107* (2015), 1–14.

[24] Baškarada, S., Nguyen, V., and Koronios, A. Architecting microservices: Practical opportunities and challenges. *Journal of Computer Information Systems 60* (9 2020), 428–436.

[25] BenIdris, M. Investigate, identify and estimate the technical debt: a systematic mapping study. *International Journal of Software Engineering and Applications (IJSEA) 9*, 5 (2020).

[26] BenIdris, M., Ammar, H., and Dzielski, D. The technical debt density over multiple releases and the refactoring story. *International Journal of Software Engineering and Knowledge Engineering 31*, 01 (2021), 99–116.

[27] Berner, S., Weber, R., and Keller, R. K. Observations and lessons learned from automated testing. In *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.* (2005), pp. 571–579.

[28] Besker, T., Martini, A., and Bosch, J. The pricey bill of technical debt: When and by whom will it be paid? In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), IEEE, pp. 13–23.

[29] Besker, T., Martini, A., and Bosch, J. Time to pay up: Technical debt from a software quality perspective. In *CIbSE* (2017), pp. 235–248.

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

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

[32] Bird, C., Nagappan, N., Murphy, B., Gall, H., and Devanbu, P. Don't touch my code! examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT and the 13th European conference on Foundations of software engineering* (2011), pp. 4–14.

[33] Blessing, L. T., and Chakrabarti, A. *DRM: A design reseach methodology.* Springer, 2009.

[34] Blum, B. I. A taxonomy of software development methods. *Communications of the ACM 37*, 11 (1994), 82–94.

[35] Bosch, J. Software architecture: The next step. In *Software Architecture: First European Workshop, EWSA 2004, St Andrews, UK, May 21-22, 2004. Proceedings 1* (2004), Springer, pp. 194–199.

[36] Botsch, B. Significance and measures of association, scopes and methods of political science, 2011.

[37] Bourque, P., Fairley, R. E., et al. *Guide to the software engineering body of knowledge (SWEBOK (R)): Version 3.0*. IEEE Computer Society Press, 2014.

[38] Boylan, J. E., Goodwin, P., Mohammadipour, M., and Syntetos, A. A. Reproducibility in forecasting research. *International Journal of Forecasting 31*, 1 (2015), 79–90.

[39] Branco, A., Cohen, K. B., Vossen, P., Ide, N., and Calzolari, N. Replicability and reproducibility of research results for human language technology: Introducing an lre special section, 2017.

[40] Britto, R., Wohlin, C., and Mendes, E. An extended global software engineering taxonomy. *Journal of Software Engineering Research and Development 4*, 1 (2016), 1–24.

[41] Brooks, F. P. The mythical man-month. *Datamation 20*, 12 (1974), 44–52.

[42] Broughton, V. *Essential classification*. Facet Publishing, 2015.

[43] 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. In *Proceedings of the FSE/SDP workshop on Future of software engineering research* (2010), pp. 47–52.

[44] Brown, W. J., Malveau, R. C., Mowbray, T. J., and Wiley, J. *AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis*, vol. 3. Wiley, 1998.

[45] Broy, M. A logical approach to systems engineering artifacts: semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views. *Software & Systems Modeling 17*, 2 (2018), 365–393.

[46] Casale, G., Chesta, C., Deussen, P., Di Nitto, E., Gouvas, P., Koussouris, S., Stankovski, V., Symeonidis, A., Vlassiou, V., Zafeiropoulos, A., et al. Current and future challenges of software engineering for services and applications. *Procedia computer science 97* (2016), 34–42.

[47] Chapin, N., Hale, J. E., Khan, K. M., Ramil, J. F., and Tan, W.-G. Types of software evolution and software maintenance. *Journal of software maintenance and evolution: Research and Practice 13*, 1 (2001), 3–30.

[48] Chatzigeorgiou, A., and Manakos, A. Investigating the evolution of bad smells in object-oriented code. In *2010 Seventh International Conference on the Quality of Information and Communications Technology* (2010), IEEE, pp. 106–115.

[49] Cicchetti, A., Borg, M., Sentilles, S., Wnuk, K., Carlson, J., and Papatheocharous, E. Towards software assets origin selection supported by a knowledge repository. In *2016 1st International Workshop on Decision Making in Software ARCHitecture (MARCH)* (2016), IEEE, pp. 22–29.

[50] Codabux, Z., and Williams, B. Managing technical debt: An industrial case study. In *2013 4th International Workshop on Managing Technical Debt (MTD)* (2013), IEEE, pp. 8–15.

[51] Coghlan, D., and Brannick, T. *Doing Action Research in Your Own Organization (4th ed.)*. London: Sage, 2014.

[52] Constantopoulos, P., and Doerr, M. Component classification in the software information base. *Object-Oriented Software Composition* (1995), 177.

[53] Conway, M. E. How do committees invent. *Datamation 14*, 4 (1968), 28–31.

[54] Creswell, J. W., and Creswell, J. D. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2017.

[55] Cunningham, W. The WyCash portfolio management system. *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '92) (Addendum) 4*, 2 (apr 1993), 29–30.

[56] Curtis, B., Sappidi, J., and Szynkarski, A. Estimating the principal of an application's technical debt. *IEEE software 29*, 6 (2012), 34–42.

[57] de Bassi, P. R., Wanderley, G. M. P., Banali, P. H., and Paraiso, E. C. Measuring developers' contribution in source code using quality metrics. In *2018 IEEE 22nd International Conference on Computer Supported Cooperative Work in Design ((CSCWD))* (2018), IEEE, pp. 39–44.

[58] Diamantopoulos, T., Papamichail, M. D., Karanikiotis, T., Chatzidimitriou, K. C., and Symeonidis, A. L. Employing contribution and quality metrics for quantifying the software development process. In *Proceedings of the 17th International Conference on Mining Software Repositories* (2020), pp. 558–562.

[59] Digkas, G., Chatzigeorgiou, A. N., Ampatzoglou, A., and Avgeriou, P. C. Can clean new code reduce technical debt density. *IEEE Transactions on Software Engineering* (2020).

[60] Digkas, G., Lungu, M., Avgeriou, P., Chatzigeorgiou, A., and Ampatzoglou, A. How do developers fix issues and pay back technical debt in the Apache ecosystem? In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering* (2018), pp. 153–163.

[61] Digkas, G., Lungu, M., Chatzigeorgiou, A., and Avgeriou, P. The evolution of technical debt in the apache ecosystem. European Conference on Software Architecture, pp. 51–66.

[62] Dingsøyr, T., Faegri, T. E., and Itkonen, J. What is large in large-scale? a taxonomy of scale for agile software development. In *International Conference on Product-Focused Software Process Improvement* (2014), vol. 8892, pp. 273–276.

[63] dos Santos, P. S. M., Varella, A., Dantas, C. R., and Borges, D. B. Visualizing and managing technical debt in agile development: An experience report. In *Agile Processes in Software Engineering and Extreme Programming: 14th International Conference, XP 2013, Vienna, Austria, June 3-7, 2013. Proceedings 14* (2013), Springer, pp. 121–134.

[64] Eken, B., Palma, F., Ayşe, B., and Ayşe, T. An empirical study on the effect of community smells on bug prediction. *Software Quality Journal 29* (2021), 159–194.

[65] Ernst, N. A. On the role of requirements in understanding and managing technical debt. In *2012 Third International Workshop on Managing Technical Debt (MTD)* (2012), pp. 61–64.

[66] Felderer, M., and Travassos, G. H. The evolution of empirical methods in software engineering. In *Contemporary Empirical Methods in Software Engineering*. Springer, 2020, pp. 1–24.

[67] Felizardo, K. R., and Carver, J. C. Automating systematic literature review. *Contemporary Empirical Methods in Software Engineering* (2020), 327–355.

[68] Fernández-Sánchez, C., Garbajosa, J., Yagüe, A., and Perez, J. Identification and analysis of the elements required to manage technical debt by means of a systematic mapping study. *Journal of Systems and Software 124* (2017), 22–38.

[69] Field, A. *Discovering statistics using IBM SPSS statistics*. Sage, 2018.

[70] Flyvbjerg, B. Five misunderstandings about case-study research. *Qualitative Inquiry 12* (2006), 219–245.

[71] Forsgren, N. Octoverse spotlight: An analysis of developer productivity, work cadence, and collaboration in the early days of covid-19, 2020.

[72] Foucault, M., Falleri, J.-R., and Blanc, X. Code ownership in open-source software. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (2014), pp. 1–9.

[73] Fowler, M. Code ownership, 2006.

[74] Fowler, M. CodeSmell, 2006.

[75] Fowler, M. Microservices, 2014.

[76] Fowler, M. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[77] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. *Refactoring: Improving the Design of Existing Code*, 1st ed. Addison Wesley, 1999.

[78] Fox, M., Green, G., and Martin, P. *Doing practitioner research*. Sage, 2007.

[79] Fujiwara, K., Fushida, K., Yoshida, N., and Iida, H. Assessing refactoring instances and the maintainability benefits of them from version archives. In *International Conference on Product Focused Software Process Improvement* (2013), Springer, pp. 313–323.

[80] Garriga, M. Towards a taxonomy of microservices architectures. In *International Conference on Software Engineering and Formal Methods* (2017), Springer, pp. 203–218.

[81] Glass, R. L., and Vessey, I. Contemporary application-domain taxonomies. *IEEE Software 12*, 4 (1995), 63–76.

[82] Glass, R. L., Vessey, I., and Ramesh, V. Research in software engineering: an analysis of the literature. *Information and Software technology 44*, 8 (2002), 491–506.

[83] Gousios, G., Kalliamvakou, E., and Spinellis, D. Measuring developer contribution from software repository data. In *Proceedings of the 2008 international working conference on Mining software repositories* (2008), pp. 129–132.

[84] Greiler, M., Herzig, K., and Czerwonka, J. Code ownership and software quality: A replication study. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories* (2015), IEEE, pp. 2–12.

[85] Griffith, I., Taffahi, H., Izurieta, C., and Claudio, D. A simulation study of practical methods for technical debt management in agile software development. In *Proceedings of the Winter Simulation Conference 2014* (2014), IEEE, pp. 1014–1025.

[86] Grubb, P., and Takang, A. A. *Software maintenance: concepts and practice*. World Scientific, 2003.

[87] Guaman, D., Sarmiento, P., Barba-Guamán, L., Cabrera, P., and Enciso, L. Sonarqube as a tool to identify software metrics and technical debt in the source code through static analysis. In *7th International Workshop on Computer Science and Engineering, WCSE* (2017), pp. 171–175.

[88] Hamill, P. *Unit test frameworks: tools for high-quality software development*. " O'Reilly Media, Inc.", 2004.

[89] Idowu, S., Strüber, D., and Berger, T. Asset management in machine learning: State-of-research and state-of-practice. *ACM Comput. Surv.* (jun 2022).

[90] ISO/IEC. IEC 27005:2018 Information technology–security techniques–information security risk management. Tech. Rep. 0, 2018.

[91] ISO/IEC/IEEE. Systems and software engineering. ISO/IEC/IEEE 24765:2010. Tech. rep., ISO/IEC/IEEE, 2010.

[92] ISO/IEC/IEEE. Asset management - Overview, principles, andsch terminology ISO/IEC/IEEE 55000:2014. Tech. rep., ISO/IEC/IEEE, 2014.

[93] Ivarsson, M., and Gorschek, T. A method for evaluating rigor and industrial relevance of technology evaluations. *Empirical Software Engineering 16*, 3 (2011), 365–395.

[94] Jamshidi, P., Pahl, C., Mendonça, N. C., Lewis, J., and Tilkov, S. Microservices: The journey so far and challenges ahead. *IEEE Software 35* (2018), 24–35.

[95] Kamola, M. How to verify conway's law for open source projects. *IEEE Access 7* (2019), 38469–38480.

[96] Kaplan, E. L., and Meier, P. Nonparametric estimation from incomplete observations. *Journal of the American statistical association 53*, 282 (1958), 457–481.

[97] Kernighan, B. W. 'programming in c- a tutorial. *Unpublished internal memorandum, Bell Laboratories* (1974).

[98] Khurum, M., Gorschek, T., and Wilson, M. The software value map—an exhaustive collection of value aspects for the development of software intensive products. *Journal of software: Evolution and Process 25*, 7 (2013), 711–741.

[99] Kim, M., Zimmermann, T., and Nagappan, N. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12* (2012), Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering - FSE '12, ACM Press, p. 1.

[100] Kitchenham, B., Brereton, O. P., Budgen, D., Turner, M., Bailey, J., and Linkman, S. Systematic literature reviews in software engineering–a systematic literature review. *Information and software technology 51*, 1 (2009), 7–15.

[101] Klotins, E., and Gorschek, T. Continuous software engineering in the wild. In *Software Quality: The Next Big Thing in Software Engineering and Quality* (Cham, 2022), Springer International Publishing, pp. 3–12.

[102] Klotins, E., Unterkalmsteiner, M., and Gorschek, T. Software-intensive product engineering in start-ups: a taxonomy. *IEEE Software 35*, 4 (2018), 44–52.

[103] Kroll, P., and Kruchten, P. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP: A Practitioner's Guide to the RUP*. Addison-Wesley Professional, 2003.

[104] Kruchten, P. The rational unified process 2nd edition: An introduction, 2000.

[105] Kruchten, P. *The rational unified process: an introduction*. Addison-Wesley Professional, 2004.

[106] Kruchten, P., Nord, R., and Ozkaya, I. *Managing Technical Debt: Reducing Friction in Software Development*. Addison-Wesley Professional, 2019.

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

[108] Kruchten, P. B. The 4+ 1 view model of architecture. *IEEE software 12*, 6 (1995), 42–50.

[109] Kujala, S., and Miron-Shatz, T. Emotions, experiences and usability in real-life mobile phone use. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (2013), pp. 1061–1070.

[110] Kumar, A., Nori, K. V., Natarajan, S., and Lokku, D. S. Value matrix: From value to quality and architecture. In *Economics-Driven Software Architecture*. Elsevier, 2014, pp. 205–240.

[111] Kwasnik, B. H. The role of classification structures in reflecting and building theory. *Advances in Classification Research Online 3*, 1 (1992), 63–82.

[112] Larivière, V., Pontille, D., and Sugimoto, C. R. Investigating the division of scientific labor using the contributor roles taxonomy (credit). *Quantitative Science Studies 2*, 1 (2021), 111–128.

[113] Law, E. L.-C., and Van Schaik, P. Modelling user experience–an agenda for research and practice. *Interacting with computers 22*, 5 (2010), 313–322.

[114] Lehman, M. On understanding laws, evolution, and conservation in the large-program life cycle. *Journal of Systems and Software 1* (1 1979), 213–221.

[115] Lehman, M. M. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE 68* (1980), 1060–1076.

[116] Lehman, M. M. Laws of software evolution revisited. In *Software Process Technology: 5th European Workshop, EWSPT'96 Nancy, France, October 9–11, 1996 Proceedings 5* (1996), Springer, pp. 108–124.

[117] Lehmann, E. L., and D'Abrera, H. J. *Nonparametrics: statistical methods based on ranks.* Holden-day, 1975.

[118] 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), 110827.

[119] Lenarduzzi, V., and Fucci, D. Towards a holistic definition of requirements debt. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement* (2019), pp. 1–5.

[120] Lenarduzzi, V., Lomio, F., Huttunen, H., and Taibi, D. Are sonarqube rules inducing bugs? In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2020), IEEE, pp. 501–511.

[121] 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* (4 2023), 111575.

[122] Lenarduzzi, V., Saarimaki, N., and Taibi, D. On the diffuseness of code technical debt in java projects of the apache ecosystem. In *2019 IEEE/ACM international conference on technical debt (TechDebt)* (2019), IEEE, pp. 98–107.

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

[124] Leung, L. Validity, reliability, and generalizability in qualitative research. *Journal of family medicine and primary care 4*, 3 (2015), 324.

[125] Levén, W., Broman, H., Besker, T., and Torkar, R. The broken windows theory applies to technical debt. *arXiv preprint arXiv:2209.01549* (2022).

[126] Leys, C., Ley, C., Klein, O., Bernard, P., and Licata, L. Detecting outliers: Do not use standard deviation around the mean, use absolute deviation around the median. *Journal of Experimental Social Psychology 49*, 4 (2013), 764–766.

[127] Li, B., Vendome, C., Linares-Vásquez, M., Poshyvanyk, D., and Kraft, N. A. Automatically documenting unit test cases. In *2016 IEEE international conference on software testing, verification and validation (ICST)* (2016), IEEE, pp. 341–352.

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

[129] Lientz, B. P., and Swanson, E. B. *Software maintenance management: A study of the maintenance of computer application software in 487 data processing organizations.* Addison-Wesley, 1980.

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

[131] Lozano, A., Wermelinger, M., and Nuseibeh, B. Assessing the impact of bad smells using historical information. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting* (2007), ACM, pp. 31–34.

[132] Mann, H. B., and Whitney, D. R. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics* (1947), 50–60.

[133] Marcilio, D., Bonifácio, R., Monteiro, E., Canedo, E., Luz, W., and Pinto, G. Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)* (2019), IEEE, pp. 209–219.

[134] Martin, R. C. *Clean Code*. Prentice Hall, 2008.

[135] 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), 42–61.

[136] Martini, A., and Bosch, J. The magnificent seven: towards a systematic estimation of technical debt interest. In *Proceedings of the XP2017 Scientific Workshops* (2017), pp. 1–5.

[137] Martini, A., Bosch, J., and Chaudron, M. Investigating architectural technical debt accumulation and refactoring over time: A multiple-case study. *Information and Software Technology 67* (2015), 237–253.

[138] Martini, A., Stray, V., and Moe, N. B. Technical-, social- and process debt in large-scale agile: An exploratory case-study. In *Agile Processes in Software Engineering and Extreme Programming – Workshops* (Cham, 2019), R. Hoda, Ed., Springer International Publishing, pp. 112–119.

[139] Mauerer, W., Joblin, M., Tamburri, D. A., Paradis, C., Kazman, R., and Apel, S. In search of socio-technical congruence: A large-scale longitudinal study. *arXiv preprint arXiv:2105.08198* (2021).

[140] McCabe, T. J. A complexity measure. *IEEE Transactions on software Engineering*, 4 (1976), 308–320.

[141] Méndez, D., Böhm, W., Vogelsang, A., Mund, J., Broy, M., Kuhrmann, M., and Weyer, T. Artefacts in software engineering: a fundamental positioning. *Software & Systems Modeling 18*, 5 (2019), 2777–2786.

[142] Méndez, D., Penzenstadler, B., Kuhrmann, M., and Broy, M. A meta model for artefact-orientation: fundamentals and lessons learned in requirements engineering. In *International Conference on Model Driven Engineering Languages and Systems* (2010), Springer, pp. 183–197.

[143] Méndez Fernández, D., Böhm, W., Vogelsang, A., Mund, J., Broy, M., Kuhrmann, M., and Weyer, T. Artefacts in software engineering: a fundamental positioning. *Software and Systems Modeling 18*, 5 (oct 2019), 2777–2786.

[144] Meneely, A., and Williams, L. Secure open source collaboration: An empirical study of linus' law. In *16th ACM Conference on Computer and Communications Security* (2009), Association for Computing Machinery, pp. 453–462.

[145] Mens, T., and Demeyer, S. *Software Evolution*. Springer Berlin Heidelberg, 2008.

[146] Menzies, T., Greenwald, J., and Frank, A. Data mining static code attributes to learn defect predictors. *IEEE transactions on software engineering 33*, 1 (2006), 2–13.

[147] Merriam, S. B., and Tisdell, E. J. *Qualitative research: A guide to design and implementation*. John Wiley & Sons, 2015.

[148] Miles, M. B., Huberman, A. M., and Saldaña, J. Qualitative data analysis: A methods sourcebook. 3rd, 2014.

[149] Miller Jr, R. G. *Survival analysis*, vol. 66. John Wiley & Sons, 2011.

[150] Munson, J. C., and Elbaum, S. G. Code churn: A measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)* (1998), IEEE, pp. 24–31.

[151] Müter, L., Deoskar, T., Mathijssen, M., Brinkkemper, S., and Dalpiaz, F. Refinement of user stories into backlog items: Linguistic structure and action verbs. In *International Working Conference*

on Requirements Engineering: Foundation for Software Quality (2019), Springer, pp. 109–116.

[152] Nagappan, M., Zimmermann, T., and Bird, C. Diversity in software engineering research. Proceedings of the 2013 9th joint meeting on foundations of software engineering, pp. 466–476.

[153] Nagappan, N., Murphy, B., and Basili, V. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering* (2008), pp. 521–530.

[154] Neamtiu, I., Xie, G., and Chen, J. Towards a better understanding of software evolution: an empirical study on open-source software. *Journal of Software: Evolution and Process 25*, 3 (2013), 193–218.

[155] Newman, S. *Building Microservices*. O'Reilly, 2015.

[156] Newman, S. *Building microservices*. O'Reilly Media, Inc., 2021.

[157] NIST. NIST Special Publication 800-30 Revision 1 - Guide for Conducting Risk Assessments. Tech. Rep. September, 2012.

[158] Nordbarg, M. E. Managing code ownership. *IEEE Software 20* (3 2003), 26–33.

[159] Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., Donohoe, P., Jones, L., Krut, R., Little, R., et al. A framework for software product line practice, version 5.0. *SEI.–2007–http://www. sei. cmu. edu/productlines/index. html* (2007).

[160] Northrop, L., Feiler, P., Gabriel, R. P., Linger, R., Longstaff, T., Kazman, R., Klein, M., Shcmidt, D., Sullivan, K., and Wallnau, K. Ultra-large-scale systems the software challenge of the future ultra-large-scale systems: The software challenge of the future. Tech. rep., Software Engineering Institute, Carnegie Mellon University, 2006.

[161] Oliveira, E., Fernandes, E., Steinmacher, I., Cristo, M., Conte, T., and Garcia, A. Code and commit metrics of developer productivity: a study on team leaders perceptions. *Empirical Software Engineering 25*, 4 (2020), 2519–2549.

[162] Oliveira, F., Goldman, A., and Santos, V. Managing technical debt in software projects using scrum: An action research. In *2015 Agile Conference* (2015), IEEE, pp. 50–59.

[163] Omair, A., et al. Sample size estimation and sampling techniques for selecting a representative sample. *Journal of Health Specialties 2*, 4 (2014), 142.

[164] Palomba, F., Bavota, G., Penta, M. D., Fasano, F., Oliveto, R., and Lucia, A. D. On the diffuseness and the impact on maintainability of code smells: a large scale empirical investigation. *Empirical Software Engineering 23*, 3 (2018), 1188–1221.

[165] Palomba, F., Bavota, G., Penta, M. D., Oliveto, R., and Lucia, A. D. Do They Really Smell Bad? A Study on Developers' Perception of Bad Code Smells. In *2014 IEEE International Conference on Software Maintenance and Evolution* (sep 2014), IEEE, pp. 101–110.

[166] Palomba, F., Zaidman, A., Oliveto, R., and De Lucia, A. An Exploratory Study on the Relationship between Changes and Refactoring. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (may 2017), IEEE, pp. 176–185.

[167] Pantiuchina, J., Lanza, M., and Bavota, G. Improving code: The (mis) perception of quality metrics. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2018), IEEE, pp. 80–91.

[168] Parizi, R. M., Spoletini, P., and Singh, A. Measuring team members' contributions in software engineering projects using git-driven technology. In *2018 IEEE Frontiers in Education Conference (FIE)* (2018), IEEE, pp. 1–5.

[169] Peters, R., and Zaidman, A. Evaluating the lifespan of code smells using software repository mining. In *2012 16th European Conference on Software Maintenance and Reengineering* (2012), IEEE, pp. 411–416.

[170] Posnett, D., D'Souza, R., Devanbu, P., and Filkov, V. Dual ecological measures of focus in software development. In *2013 35th International Conference on Software Engineering (ICSE)* (2013), IEEE, pp. 452–461.

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

[172] Rahman, F., and Devanbu, P. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), pp. 491–500.

[173] Ralph, P., Baltes, S., Adisaputri, G., Torkar, R., Kovalenko, V., Kalinowski, M., Novielli, N., Yoo, S., Devroey, X., Tan, X., et al. Pandemic programming: how covid-19 affects software developers and how their organizations can help. *arXiv preprint arXiv:2005.01127* (2020).

[174] Ralph, P., and Tempero, E. Construct validity in software engineering research and software metrics. In *22nd International Conference on Evaluation and Assessment in Software Engineering* (Christchurch, New Zealand, 2018), Association for Computing Machinery (ACM).

[175] Rapu, D., Ducasse, S., Gîrba, T., and Marinescu, R. Using history information to improve design flaws detection. In *Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.* (2004), IEEE, pp. 223–232.

[176] Raymond, E. The cathedral and the bazaar. *Knowledge, Technology & Policy 12* (1999), 23–49.

[177] Reussner, R., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J., and Märtin, L. *Managed software evolution*. Springer Nature, 2019.

[178] Rios, N., de Mendonça Neto, M. G., 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), 117–145.

[179] Robson, C. *Real world research*, vol. 3. Wiley Chichester, 2011.

[180] Runeson, P., and Höst, M. Guidelines for conducting and reporting case study research in software engineering. *Empirical software engineering 14*, 2 (2009), 131–164.

[181] Runeson, P., Host, M., Rainer, A., and Regnell, B. *Case Study Research in Software Engineering: Guidelines and Examples*. John Wiley and Sons, 2012.

[182] Saarimäki, N., Lenarduzzi, V., and Taibi, D. On the diffuseness of code technical debt in java projects of the apache ecosystem. In *Proceedings of the Second International Conference on Technical Debt* (2019), IEEE Press, pp. 98–107.

[183] Saher, N., Baharom, F., and Ghazali, O. Requirement change taxonomy and categorization in agile software development. In *2017 6th International Conference on Electrical Engineering and Informatics (ICEEI)* (2017), IEEE, pp. 1–6.

[184] Saldaña, J. *The coding manual for qualitative researchers*. Sage, 2015.

[185] Santos, J. A. M., Rocha-Junior, J. B., Prates, L. C. L., do Nascimento, R. S., Freitas, M. F., and de Mendonça, M. G. A systematic review on the code smell effect. *Journal of Systems and Software 144*, July (oct 2018), 450–477.

[186] Schneider, J., Gaul, A. J., Neumann, C., Hogräfer, J., Wellssow, W., Schwan, M., and Schnettler, A. Asset management techniques. *International Journal of Electrical Power & Energy Systems 28*, 9 (2006), 643–654.

[187] Schweik, C. M., English, R. C., Kitsing, M., and Haire, S. Brooks' versus linus' law: An empirical test of open source projects. In *9th Annual International Digital Government Research Conference* (2008), pp. 423–424.

[188] Siavvas, M., Tsoukalas, D., Jankovic, M., Kehagias, D., and Tzovaras, D. Technical debt as an indicator of software security risk: a machine learning approach for software development enterprises. *Enterprise Information Systems 16*, 5 (2022), 1824017.

[189] Silva, D., Tsantalis, N., and Valente, M. T. Why we refactor? confessions of GitHub contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2016* (New York, New York, USA, 2016), ACM Press, pp. 858–870.

[190] Silva, M., Oliveira, T., and Bastos, R. Software artifact metamodel. In *2009 XXIII Brazilian Symposium on Software Engineering* (2009), pp. 176–186.

[191] Sjøberg, D. I., and Bergersen, G. R. Construct validity in software engineering. *IEEE Transactions on Software Engineering* (2022), 1–1.

[192] Sjøberg, D. I., Yamashita, A., Anda, B. C., Mockus, A., and Dybå, T. Quantifying the effect of code smells on maintenance effort. *IEEE Transactions on Software Engineering 39*, 8 (2012), 1144–1156.

[193] Šmite, D., Brede Moe, N., Klotins, E., and Gonzalez-Huerta, J. From forced working-from-home to voluntary working-from-anywhere: Two revolutions in telework. *Journal of Systems and Software 195* (2023), 111509.

[194] Šmite, D., Wohlin, C., Galviņa, Z., and Prikladnicki, R. An empirically based terminology and taxonomy for global software engineering. *Empirical Software Engineering 19*, 1 (2014), 105–153.

[195] Sommerville, I. Software engineering. 10th. In *Book Software Engineering. 10th, Series Software Engineering*. Addison-Wesley, 2015.

[196] Stol, K.-J., and Fitzgerald, B. The abc of software engineering research. *ACM Transactions on Software Engineering and Methodology (TOSEM) 27*, 3 (2018), 1–51.

[197] Stringer, E. T. *Action Research (4th Edition)*. Thousand Oaks, CA: Sage, 2014.

[198] Sundelin, A., Gonzalez-Huerta, J., and Wnuk, K. The hidden cost of backward compatibility: When deprecation turns into technical debt - An experience report. In *Proceedings - 2020 IEEE/ACM International Conference on Technical Debt, TechDebt 2020* (2020), pp. 67–76.

[199] Sundelin, A., Gonzalez-Huerta, J., Wnuk, K., and Gorschek, T. Towards an anatomy of software craftsmanship. *ACM Transactions on Software Engineering and Methodology (TOSEM) 31*, 1 (2021), 1–49.

[200] Suryanarayana, G., Samarthyam, G., and Sharma, T. *Refactoring for Software Design Smells: Managing Technical Debt*, vol. 11. Elsevier Science, 2014.

[201] Svahnberg, M., Van Gurp, J., and Bosch, J. A taxonomy of variability realization techniques. *Software: Practice and experience 35*, 8 (2005), 705–754.

[202] Taivalsaari, A., and Mikkonen, T. A taxonomy of iot client architectures. *IEEE software 35*, 3 (2018), 83–88.

[203] Thongtanunam, P., McIntosh, S., Hassan, A. E., and Iida, H. Revisiting code ownership and its relationship with software quality in the scope of modern code review. In *Proceedings of the 38th international conference on software engineering* (2016), pp. 1039–1050.

[204] Tilley, S., and Huang, S. Documenting software systems with views iii: towards a task-oriented classification of program visualization techniques. In *Proceedings of the 20th annual international conference on Computer documentation* (2002), pp. 226–233.

[205] Tofan, D., Galster, M., and Avgeriou, P. Reducing architectural knowledge vaporization by applying the repertory grid technique. In *Software Architecture: 5th European Conference, ECSA 2011, Essen, Germany, September 13-16, 2011. Proceedings 5* (2011), Springer, pp. 244–251.

[206] Tom, E., Aurum, A., and Vidgen, R. A consolidated understanding of technical debt. In *ECIS 2012 Proceedings* (2012).

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

[208] Tornhill, A. *Your Code As A Crime Scene*. The Pragmatic Bookshelf, 2015.

[209] Tornhill, A. *Software Design X-Rays: Fix Technical Debt with Behavioral Code Analysis*. Pragmatic Bookshelf, 2018.

[210] Tornhill, A., and Borg, M. Code red: the business impact of code quality-a quantitative study of 39 proprietary production codebases. In *Proceedings of the International Conference on Technical Debt* (2022), pp. 11–20.

[211] Tsantalis, N., Mansouri, M., Eshkevari, L. M., Mazinanian, D., and Dig, D. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on SoftwarTe Engineering* (2018), ACM, pp. 483–494.

[212] Tsoukalas, D., Kehagias, D., Siavvas, M., and Chatzigeorgiou, A. Technical debt forecasting: an empirical study on open-source repositories. *Journal of Systems and Software 170* (2020), 110777.

[213] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. When and why your code starts to smell bad. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1* (2015), IEEE Press, pp. 403–414.

[214] Tufano, M., Palomba, F., Bavota, G., Oliveto, R., Di Penta, M., De Lucia, A., and Poshyvanyk, D. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering 43*, 11 (2017), 1063–1088.

[215] Unterkalmsteiner, M., Feldt, R., and Gorschek, T. A taxonomy for requirements engineering and software test alignment. *ACM Transactions on Software Engineering and Methodology (TOSEM) 23*, 2 (2014), 1–38.

[216] Usman, M., Britto, R., Börstler, J., and Mendes, E. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology 85* (2017), 43–59.

[217] Vassallo, C., Panichella, S., Palomba, F., Proksch, S., Zaidman, A., and Gall, H. C. Context is king: The developer perspective on the usage of static analysis tools. In *25th IEEE International Conference on Software Analysis, Evolution and Reengineering* (2018).

[218] Vetro, A. Using automatic static analysis to identify technical debt. In *2012 34th International Conference on Software Engineering (ICSE)* (2012), IEEE, pp. 1613–1615.

[219] von Linné, C. *Systema naturae; sive, Regna tria naturae: systematice proposita per classes, ordines, genera & species.* Haak, 1735.

[220] Wohlin, C. Guidelines for snowballing in systematic literature studies and a replication in software engineering. In *Proceedings of the 18th international conference on evaluation and assessment in software engineering* (2014), pp. 1–10.

[221] Wohlin, C., Wnuk, K., Šmite, D., Franke, U., Badampudi, D., and Cicchetti, A. Supporting strategic decision-making for selection of software assets. In *International conference of software business* (2016), Springer, pp. 1–15.

[222] Wolfram, K., Martine, M., and Sudnik, P. Recognising the types of software assets and its impact on asset reuse. In *European Conference on Software Process Improvement* (2020), Springer, pp. 162–174.

[223] Yamashita, A., and Moonen, L. Do code smells reflect important maintainability aspects? In *2012 28th IEEE international conference on software maintenance (ICSM)* (2012), IEEE, pp. 306–315.

[224] Yamashita, A., and Moonen, L. Exploring the impact of inter-smell relations on software maintainability: An empirical study. In *Proceedings of the 2013 International Conference on Software Engineering* (2013), IEEE Press, pp. 682–691.

[225] Yin, R. K. *Case study research and applications: Design and Methods*, 6 ed. Sage, 2018.

[226] 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), 195–218.

[227] Yoshida, N., Saika, T., Choi, E., Ouni, A., and Inoue, K. Revisiting the relationship between code smells and refactoring. In *2016 IEEE 24th International Conference on Program Comprehension (ICPC)* (may 2016), vol. 2016-July, IEEE, pp. 1–4.

[228] Zabardast, E., Frattini, J., Gonzalez-Huerta, J., Mendez, D., Gorschek, T., and Wnuk, K. Assets in software engineering: What are they after all? *Journal of Systems and Software 193* (2022), 111485.

[229] Zabardast, E., Gonzalez-Huerta, J., and Palma, F. The impact of forced working-from-home on code technical debt: An industrial case study. In *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2022), IEEE, pp. 298–305.

[230] Zabardast, E., Gonzalez-Huerta, J., and Šmite, D. Refactoring, bug fixing, and new development effect on technical debt: An industrial case study. *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (2020), 376–384.

[231] Zabardast, E., Gonzalez-Huerta, J., and Tanveer, B. Ownership vs contribution: Investigating the alignment between ownership and contribution. In *19th IEEE International Conference on Software Architectures* (2022), IEEE.

[232] Zhao, Y., Dong, J., and Peng, T. Ontology classification for semantic-web-based software engineering. *IEEE Transactions on Services Computing 2*, 4 (2009), 303–317.

ABSTRACT

Background: As software is everywhere, and almost every company has nowadays a dependency on software, designing and developing software-intensive products or services has become significantly challenging and time-consuming. The challenges are due to the continuous growth of the size and complexity of software and the fast pace of change. It is important that software-developing organisations' engineering practises adapt to the rising challenges by adopting well-engineered development activities.

Organisations deal with many software artefacts, some of which are more relevant for the organisation. We define Software Assets as artefacts intended to be used more than once. Given software development's continuous and evolutionary aspect, the assets involved degrade over time. Organisations need to understand what assets are relevant and how they degrade to exercise quality control over software assets. Asset degradation is inevitable, and it may manifest in different ways.

Objective: The main objective of this thesis is: (i) to contribute to the software engineering body of knowledge by providing an understanding of what assets are and how they degrade; and (ii) to gather empirical evidence regarding asset degradation and different factors that might impact it on industrial settings.

Method: To achieve the thesis goals, several studies have been conducted. The collected data is from peer-reviewed literature and collaboration with five companies that included extracting archival data from over 20 million LOC and archival data from open-source repositories.

Results: The first contribution of this thesis is defining the concept of assets and asset degradation in a position paper. We aim to provide an understanding of software assets and asset degradation and its impact on software development.

Additionally, a taxonomy of assets is created using academic and industrial input. The taxonomy includes 57 assets and their categories. To further investigate the concept of asset degradation, we have conducted in-depth analyses of multiple industrial case studies on selected assets. This thesis presents results to provide evidence on the impact of different factors on asset degradation, including: (i) how the accumulation of technical debt is affected by different development activities; (ii) how degradation 'survives'; and (iii) how working from home or the misalignment between ownership and contribution impacts the faster accumulation of asset degradation. Additionally, we created a model to calculate the degree of the alignment between ownership and contribution to code.

Conclusion: The results can help organisations identify and understand the relevant software assets and characterise their quality degradation. Understanding how assets degrade and which factors might impact their faster accumulation is the first step to conducting sufficient and practical asset management activities. For example, by engaging (i) proactively in preventing uncontrolled growth of degradation (e.g., aligning ownership and contribution); and (ii) reactively in prioritising mitigation strategies and activities (focusing on recently introducing TD items).

BLEKINGE TEKNISKA HÖGSKOLA · BTH ·

ISSN 1653-2090

2023:05

ISBN 978-91-7295-455-7