Technical Debt Prioritization

Methods, Techniques, and a Large Exploratory Study

Diogo de Jesus Pina

Thesis presented to the
Institute of Mathematics and Statistics
of the University of São Paulo
in partial fulfillment
of the requirements
for the degree of
Doctor of Science

Program: Ciência da Computação
Advisor: Prof. Dr. Alfredo Goldman

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Finance Code 001

São Paulo
August, 2023

Technical Debt Prioritization

Methods, Techniques, and a Large Exploratory Study

Diogo de Jesus Pina

Special Thanks

For from him and through him and for him
are all things. To him be glory forever. Amen.
— Romans XI, 36

First, I want to thank God for allowing me to fulfill this dream with a lot of dedication, opportunities, and sacrifices. I also want to thank God for all the miracles he worked during my life, especially during these years of graduation. Miracles ranged from simple gestures of kindness, materializing dreams, and my son's life.

I thank my wife, Luana, and my son, John, for all the support, love, and understanding for my absence so that I could study and carry out the research. I thank my parents, Ramiro and Sonia, and my godmother Francisca, for all the support from basic education to the present day.

I want to thank Professor Dr. Alfredo Goldman, who has supported and guided me in my studies and academic life since my undergraduate. I also want to thank Dr. Carolyn Seaman, who allowed me to study at UMBC, helping to enrich my academic and cultural knowledge. Both professors were very important; without them, I could not have developed high-quality research.

I also thank my American family, especially Susie and Michael, for supporting us during the sandwich period. I also thank my Brazilian family and friends who supported me, especially my mother-in-law Elizabete.

Last but not least special, I thank the "ship's crew" for all the guidance and advice in my work and my life.

Resumo

Diogo de Jesus Pina. **Priorização de Dívida Técnica:** ***Métodos, Técnicas e um Estudo Exploratório***. Tese (Doutorado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Equipes de desenvolvimento de software precisam priorizar o pagamento de itens de dívida técnica para melhorar a qualidade do software e garantir um ritmo no desenvolvimento de novas funções e manutenção do código. Ferramentas de identificação são capazes de encontrar milhares de itens de dívida técnica de código em um repositório. Logo, é inviável pagar todos os itens, pois levaria meses ou até anos. Portanto, o time precisa decidir quais itens deveram ser pagos e quando realizar o pagamento.

Nós realizamos um mapeamento da literatura para identificar os trabalhos realizados para ajudar no processo de priorização de dívida técnica. Nós encontramos trabalhos que conceituam o processo, desenvolvem arcabouços de priorização e aplicação de diversos métodos para realizar a priorização. Apesar dos esforços realizados, ainda não foi desenvolvido um método de priorização que considera o contexto do desenvolvimento do software, funcione em várias linguagens de programação, cubram diversos tipos de dívida técnica e seja integrado a uma ferramenta para aplicá-lo na prática.

A partir do mapeamento, a nossa motivação para esta pesquisa é entender como os desenvolvedores priorizam itens de dívida técnica em projetos reais de software. Além disso, nós também aplicamos métodos de aprendizado de máquina para automatizar o processo de priorização.

Nós desenvolvemos a ferramenta Sonarlizer Xplorer para minerar e analisar projetos públicos hospedados no GitHub suportando nossos estudos. O resultado da aplicação da ferramenta é uma lista com itens de dívida técnica e métricas de código de um grande número de projetos de software.

Nós aplicamos um questionário para coletar dados de projetos Java públicos para entender quais critérios os desenvolvedores de software usam para priorizar dívida técnica de código em projetos reais. Então, analisamos os dados usando Teoria Fundamentada Straussiana e agrupamos os critérios em quinze categorias, dividindo-as em duas super-categorias relacionadas ao pagamento da dívida técnica e três relacionadas ao não pagamento. Nós encontramos que quando os desenvolvedores decidiram pagar um item de dívida técnica, eles querem pagar logo. Quando eles decidem não pagar, geralmente é porque a dívida foi adquirida intencionalmente e está relacionado a decisões de projeto. Quando eles usaram critérios parecidos, a níveis de prioridade de pagamento são parecidos. Por fim, nós observamos que cada projeto de software precisa de regras próprias para identificar seus itens de dívida técnica.

Nós também estudamos a aplicação de métodos de aprendizado de máquina para priorizar os itens de dívida técnica em projetos reais de software. Nós aplicamos o mesmo questionário do estudo anterior e obtivemos 2.616 respostas. Com as respostas, criamos um dataset usando três estratégias de rotulação: "pagar ou não", 3-classes e prioridade. Então, aplicamos nove métodos de machine learning bem-conhecidos sobre 27 métricas de código para construir um modelo para decidir se um item de dívida técnica deve ser pago (com acurácia de 0,79 e F1 de 0,85) e quando realizar o pagamento, aplicando quatro abordagens atingindo desempenho de acurácia de 0,57 usando análise tradicional e 0,81 usando análise tunada.

Palavras-chave: Dívida Técnica. Priorização de Dívida Técnica. Gerenciamento de Dívida Técnica. Aprendizado de Máquina. Inteligência Artificial.

Abstract

Diogo de Jesus Pina. **Technical Debt Prioritization: *Methods, Techniques, and a Large Exploratory Study***. Thesis (Doctorate). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

Software development teams need to prioritize the technical debt items payment to improve the software quality and ensure the new feature and code maintenance development pace. Identification tools can find thousands of technical debt items in a code repository. Thus, it is infeasible to pay off all items because it would take months or even years. Therefore, the team must decide which items should be paid off and when to pay them.

We performed a mapping review to identify studies that assist in the technical debt prioritization process. We found papers that conceptualized the process, developed prioritization frameworks, and applied various methods to accomplish prioritization. Despite the efforts, a prioritization method that considers the software development context, works for several programming languages, covers different types of technical debt, and is integrated into a tool to apply it in practice still needs to be developed.

Based on the mapping review, our motivation for this research is to understand how developers prioritize technical debt items in real software projects. Furthermore, we also apply machine learning methods to automate the prioritization process.

We developed the Sonarlizer Xplorer tool to mine and analyze public projects hosted on GitHub supporting our studies. The result of applying the tool is a list of technical debt items and code metrics for many software projects.

We applied a questionnaire to collect data from public Java projects to understand which criteria software developers use to prioritize code technical debt in real projects. We analyzed the data using Straussian Grounded Theory. We grouped the criteria into fifteen categories and divided them into two super-categories related to technical debt payment and three related to non-payment. We have found that when developers decide to pay off a technical debt item, they want to pay it off soon. When they decide not to pay, it is usually because the debt was acquired intentionally and is related to design decisions. When they used similar criteria, the payment priority levels were similar. Finally, we note that each software project needs its specific rules to identify its technical debt items.

We also study the application of machine learning methods to prioritize technical debt items in real software projects. We applied the same questionnaire as in the previous study and obtained 2,616 responses. We create a dataset using three labeling strategies: "pay or not", 3-classes, and priority. We applied nine well-known machine learning methods on 27 code metrics to build a model for deciding whether a technical debt item should be paid (with an accuracy mean of 0.79 and F1 mean of around 0.86) and when to pay, applying four approaches achieving accuracy performance of 0.57 using traditional analysis and 0.81 using tuned analysis.

Keywords: Technical Debt. Technical Debt Prioritization. Technical Debt Management. Machine Learning. Artificial Intelligence.

List of abbreviations

TD	Technical Debt
TDI	Technical Debt Item
TDM	Technical Debt Management
TDP	Technical Debt Prioritization
ML	Machine Learning
AI	Artificial Intelligence
IME	Instituto de Matemática e Estatística
USP	Universidade de São Paulo

List of Figures

2.1 Technical Debt Quadrant. (Fowler, 2009) 9
2.2 Framework to manage technical debt. (Seaman and Yuepo Guo, 2011) . . 11
2.3 Table of grades and colors of the SQALE rate. (Letouzey, 2012a) 18
2.4 Kiviat Graph for representation of technical debt by characteristics proposed by the SQALE method. (Letouzey, 2012a) 19
2.5 Technical debt pyramid of the SQALE methodology. (Letouzey, 2012a) . 20
2.6 Line chart to monitor technical debt over time. (Kniberg, 2013) 20

3.1 Petersen's process to perform a systematic map. (Petersen *et al.*, 2008) . 26
3.2 Distribution of selected studies over time. 27
3.3 Technical debt prioritization taxonomy 28

4.1 Sonarlizer Xplorer architecture. 40
4.2 GitHub data entities and relationships. 41
4.3 Repository document example. 43
4.4 Left: Example of technical debt items on SonarQube. Right: Example of code metrics on SonarQube. 44
4.5 Questionnaire question example. 47

5.1 Questionnaire question example. 51
5.2 Codes and categories for not paying off technical debt items. 58
5.3 Codes and categories for paying off technical debt items. 59

6.1 Data collection process. 64
6.2 Complexity and ncloc distribution for selected projects where a developer answered. 65
6.3 Survey Process Flow Statistics 68
6.4 Answers distributions by Label 69

List of Tables

2.1 Technical debt record example 15

3.1 Search results overview 27
3.2 Papers mapping into the first level taxonomy 29
3.3 Papers mapping into the second level taxonomy 31
3.4 Technical debt prioritization return example 33
3.5 Technical debt prioritization methods response types and characteristics 34

4.1 Example of project quantities mined by programming language. 44
4.2 Number of technical debt items by severity and projects divided by a non-comment line of code (ncloc). 44

6.1 Selected code metrics 67
6.2 Answer labels and count 68
6.3 Discriminative power analysis using Mann-Whitney U test 70
6.4 Selected Machine Learning methods, description, and hyperparameter values 72
6.5 To pay or not to pay machine learning performance 77
6.6 3-classes approach performance 77
6.7 3-classes approach tuned performance 78
6.8 Most common technical debt types approach performance 79
6.9 Most common technical debt types approach tuned performance 79
6.10 Simple approach performance 80
6.11 Simple approach tuned performance 80
6.12 2-layers approach performance 81
6.13 2-layers approach tuned performance 81

Contents

1 Introduction **1**

1.1 Motivation 2

1.2 Research Problem 2

1.3 Research Questions 3

1.4 Goals 4

1.4.1 General Goals 4

1.4.2 Specific Goals 4

1.5 Contributions 5

1.5.1 For Researchers 5

1.5.2 For Practitioners 5

1.6 Thesis Structure 5

2 Background **7**

2.1 The Technical Debt Metaphor 7

2.2 Technical Debt Classification 8

2.3 Technical Debt Properties 10

2.4 Technical Debt Management 10

2.4.1 Technical Debt Identification 10

2.4.2 Technical Debt Measure 14

2.4.3 Technical Debt Monitor 19

2.5 Technical Debt Prioritization 22

2.6 Chapter Summary 22

3 Systematic Mapping Review on Technical Debt Prioritization **25**

3.1 Research Methodology 25

3.1.1 Conduct Search 25

3.1.2 Screening of Papers 26

3.1.3 Data Extraction and Mapping Process 26

3.1.4 Classification Schema 28
3.2 Mapping Review Results 28
3.2.1 RQ1: What methods and techniques were proposed to prioritize technical debt? 28
3.2.2 First Level 28
3.2.3 Second Level 31
3.2.4 RQ2: What results do technical debt prioritization methods provide? 33
3.2.5 RQ3: What are the characteristics of the technical debt prioritization methods? 35
3.3 Discussion of Findings 35
3.4 Threats to Validity 36
3.5 Mapping Conclusion 37
3.6 Related Work for Technical Debt Prioritization Criteria Study 38
3.7 Related Work for Technical Debt Prioritization using Machine Learning . 38

4 Sonarlizer Xplorer and InteraSurveyTD 39
4.1 Sonarlizer Xplorer 39
4.1.1 Use Cases 39
4.1.2 Architecture, Technologies, and Implementation 40
4.1.3 GitHub Xplorer 40
4.1.4 Sonarlizer 42
4.1.5 Results 43
4.1.6 Related Tools 45
4.1.7 Future Enhancements 45
4.1.8 License 46
4.2 InteraSurveyTD 46
4.3 Conclusion 47

5 Technical Debt Prioritization Criteria 49
5.1 Research Method 50
5.1.1 Data Collection 50
5.1.2 Data Analysis 52
5.2 Results 53
5.2.1 Open Coding 53
5.2.2 Axial Coding 54
5.2.3 Selective Coding 57
5.3 Discussion of the Results 60
5.3.1 Implications for Researchers and Practitioners 60

5.3.2 Threats to Validity 61
5.4 Conclusions and Future Work 62

6 Technical Debt Prioritization using Machine Learning 63
6.1 Methodology 63
6.1.1 Data Collection 64
6.1.2 Data Preparation 68
6.1.3 Exploratory Analysis 69
6.1.4 Model Building 71
6.2 Results 76
6.2.1 RQ2.1: How effective are ML models for deciding **whether** or not a technical debt item should be paid? 76
6.2.2 RQ2.2: How effective are ML models for deciding **when** a technical debt item should be paid? 77
6.2.3 3-Classes Approach 77
6.2.4 RQ2.3: Which are the best Machine Learning algorithms to prioritize technical debt? 82
6.3 Discussion of the Results 82
6.4 Implications for Researchers and Practitioners 83
6.5 Threats to Validity 84
6.6 Conclusion and Future Work 85

7 Conclusions 87
7.1 Summary of Findings 87
7.2 Future Work 89

References 91

Chapter 1

Introduction

Technical debt is a metaphor for the financial debt. The metaphor is associated with the idea of interest for non-payment of technical debt items. More formally, technical debt is a metaphor for immature, incomplete, or inadequate artifacts in the software development lifecycle (Seaman and Yuepo Guo, 2011).

Acquiring a technical debt could bring benefits (Yuepu Guo, Rodrigo Oliveira Spínola, *et al.*, 2016) in a short time, such as reducing the time and effort to develop the tasks, resources, and budget reduction in features development and maintenance, allowing to keep track of business decisions. However, accumulating a large amount of technical debt could negatively impact the software's quality resulting in extra costs to develop new features and maintain the existing code.

Therefore, it is essential to balance the benefits of acquiring a technical debt and the cost and interest of paying off (Seaman and Yuepo Guo, 2011). In addition to identifying, measuring, and monitoring technical debt, prioritization could help software development teams to make-decision about whether and when technical debt items should be paid and in which order to make these payments.

Initially, the technical debt prioritization issue was identified during a case study about technical debt measuring methods (Pina and Goldman, 2016). In the study, technical debt items were identified using Sonar, and team members were asked to choose some items to pay off; however, the main question was what items should be chosen to pay off.

Technical debt identification tools like Sonar Qube have evolved over the last few years and can identify up to thousands of items. Therefore, due to time and budget constraints, development teams must decide which items should be paid off and in which order made the payment. We found several works on technical debt prioritization in the literature. However, many treat the problem conceptually, and those that propose methods are either for a very restricted set of technical debt types and development context or cannot be applied automatically.

The advancement of artificial intelligence in the past few years has allowed other areas to take advantage of its power to build more assertive and autonomous decision-making systems. The implementation of algorithms and methods to manipulate data and apply artificial intelligence techniques has helped developers and researchers to create and improve software tools Barstow, 1988; Harman, 2012. In special, research areas such as software engineering can also use these methods to improve the precision of results and provide insights that would be missed by traditional statistical analyses.

Machine learning is one of the most common approaches used in artificial intelligence Mitchell *et al.*, 1997; El Naqa and Murphy, 2015. It is been widely used in many application domains to create clusters and predict behavior/patterns in data sets. Most machine learning methods are generic and could result in high performance to analyze data and make decisions. That is, the same method can be applied to different types of problems just by adapting the input parameters.

In this thesis, we apply well-known machine learning methods for prioritizing technical debt items payment. That is, we apply methods to decide whether an item should be paid off and when to make the payment.

1.1 Motivation

The main motivation for this research is to understand how developers prioritize technical debt in real software projects and then to develop machine learning methods to prioritize technical debt automatically.

The technical prioritization problem is an issue faced in academia where several studies conceptualize it, create frameworks to prioritize it, and apply methods to solve it. On the other hand, prioritization is also a problem faced in the industry. Development teams acquire new technical debt items daily. Thus they need to prioritize efficiently the payment to save resources, avoiding interest payments or paying items that do not significantly improve the code quality and productivity of development teams.

A good solution to the technical debt prioritization problem may contribute significantly to the technical debt management area. The qualitative research will enable a better understanding of the problem and how developers perform prioritization. The machine learning methods will also allow researchers, managers, and development teams to decide which technical debt items to pay off and when the payment should be made.

1.2 Research Problem

The technical debt management process comprises the following steps: identification, measurement, and decision-making (Kruchten *et al.*, 2012). After identifying and measuring technical debt items, a software project could have hundreds or thousands of items to pay (Falessi and Voegele, 2015). Therefore, they must choose which technical debt items should be paid off and in which order to make the payment. If they do not prioritize the payment of the technical debt or the prioritization is not done efficiently, resources such as time and money can be wasted due to the difficulty of maintaining and implementing new features caused by the accumulation of priority items of technical debt.

For example, in a planning meeting, the software development team allocates part of the next three weeks to pay off technical debt items. Then, it uses a tool like Sonar Qube to identify technical debt items and measure the time it takes to pay off each item. The tool found hundreds of items. Due to time constraints, the team should choose only a few items to pay off. Then they will need to identify the most critical items to pay first. Therefore, they need a method to aid them in choosing which technical debt items must be paid off and deciding the payment order.

1.3 Research Questions

This thesis aims to address the following research questions:

- RQ1. How do developers prioritize technical debt?
  - RQ1.1. How do developers decide whether a code technical debt item should or should not be paid off?
  - RQ1.2 How do developers decide when a code technical debt item should be paid off?

To investigate these questions, we analyzed how developers prioritize technical debt in real software projects. We invited developers by email to answer two questions about technical debt items in projects they had contributed to. The first question is "When should the item be paid off?" (multiple-choice) with six sorted options from Immediately to Never. The second question is "Why?" (open-text field) so that they can explain the decision in the first question.

We applied Straussian Grounded Theory to analyze the data and understand the criteria developers used to prioritize technical debt items. We grouped the criteria into fifteen categories. Then, we associated them into five super-categories: two related to paying off the technical debt and three related to not paying off.

Some developers justified non-payment of the technical debt item by a specific software project decision. However, when they opted to pay the item off, they chose to pay it off soon. In addition, they chose the same pay priority or a neighboring level when they used similar prioritization criteria. Finally, we note that each software project needs its criteria to perform prioritization efficiently.

- RQ2. How to prioritize the payment of technical debt automatically?
  - RQ2.1. How effective are Machine Learning models for deciding **whether** or not a technical debt item should be paid?
  - RQ2.2. How effective are Machine Learning models for deciding **when** a technical debt item should be paid?
  - RQ2.3. Which are the best Machine Learning algorithms to prioritize technical debt?

For this second research question, we trained and assessed machine learning methods to predict whether a technical debt item should be paid off. Then, we applied the same algorithms using four approaches to classify a technical debt item according to priority.

For example, we categorized when a technical debt should be paid off into six categories: immediately, as soon as possible, in the next release, in the next few releases, when there is free time, or never.

As machine learning had yet to be applied to the technical debt prioritization problem, as (Tsoukalas, Mittas, *et al.*, 2021) and (Mauricio Aniche *et al.*, 2020), we decided to apply simple methods to better understand the results. The nine selected methods are Dummy Classifier, Naive Bayes, K-Nearest Neighbors, Logistic Regression, Ridge Classifier, Support Vector Machine, Decision Tree, Random Forest, and XGBoost. We applied the methods over 27 features that describe the source code, such as the number of lines, source file complexity, and the number of statements.

In all tested approaches, at least one of the methods achieved statistically superior performance compared to random choice. Specifically, for determining whether a technical debt item should be paid, they reached an accuracy of 0.86 to decide whether a technical debt item should be paid off using traditional analysis. When it came to determining when an item should be paid off, the accuracy reached 0.96 using tuned analysis. K-Nearest Neighbors and Random Forest consistently performed exceptionally well across all of the approaches, while Decision Tree and XGBoost also showed strong performance in several cases.

1.4 Goals

Below we will list the general and specific goals that will guide this research project throughout its development.

1.4.1 General Goals

This study aims to understand how developers prioritize technical debt and develop methods to aid that decision. We trained machine learning methods to decide whether a technical debt item should be paid off and when the payment should be made.

1.4.2 Specific Goals

Below is a list of the main specific objectives:

1. Identify studies on technical debt prioritization methods.
2. Develop or improve a method to identify whether a technical debt item should or should not be paid off.
3. Improve the how-to classify technical debt items by severity levels.
4. Elaborate the methods of prioritizing the order of payment of technical debt items.
5. Ensure that all methods are computer-assisted, requiring just receiving code metrics as training data.
6. Evaluate the methods' performance for each approach.

1.5 Contributions

In the subsequent subsections, we will outline the contributions that this research project aims to make for both researchers and practitioners in the field.

1.5.1 For Researchers

The developed technical debt prioritization criteria serve as a base to investigate more standards for other programming languages, kinds of projects, and paradigms. They also can use the criteria to create guidelines to assist developers in deciding the payment priority level for each technical debt item they identify in their software. In addition, researchers can relate the criteria to software context, such as code metrics and commit history, to automatically categorize payment of code technical debt.

On the other hand, the machine learning methods that were trained in this study and performed well could be used to prioritize the technical debt items as input to other studies, for example, for management or decision-making research. The methods also can be added as part of prioritization or management frameworks. They also can be a baseline for developing more complex strategies to prioritize technical debt.

1.5.2 For Practitioners

Practitioners can use the prioritization criteria to pragmatically evaluate and plan technical debt payments in their software projects. For each criterion, they can verify and decide about its application to classify the payment priority level in their project. In addition, they can implement a criteria list in a technical debt management tool to help in the decision-making process.

On the other hand, the results could assist practitioners in deciding which technical debt items should be paid off and when the payment should be made. Prioritization will allow items that caused the most interest and negative impact to be paid off first, improving the code quality and speeding up the development and maintenance.

1.6 Thesis Structure

This thesis is divided as follows:

- In Chapter 2, we provide the technical debt metaphor concept, classification, properties, management, and prioritization. We also provide the concepts for machine learning.
- In Chapter 3, we present the results of a mapping study on technical debt prioritization we performed. We develop a taxonomy of technical debt literature divided into two levels. We classify the prioritization method return into Boolean, Categories, or Ordered List. We also classify the methods according to context-adaptative, free-language, variety of technical debt types, and implementation in a tool.

- In Chapter 4, we describe Sonarlizer Xplorer and InteraSurveyTD tools. Sonarlizer Xplorer was developed to mine and analyze public software projects hosted at GitHub, resulting in a technical debt items and code metrics dataset. We also developed InteraSurveyTD to apply a survey where the developers only see technical debt items from projects they have worked on. The tool asks them whether/when a technical debt should be paid off and why.
- In Chapter 5, we present a qualitative study that aims to understand which criteria software developers use in practice to prioritize code technical debt items in real software projects. We applied a survey to collect developers' answers. We analyzed the answers using Straussian Grounded Theory (Straussian GT) techniques, namely open coding, axial coding, and selective coding. We grouped the criteria into 15 categories, 2 super-categories related to paying off the technical debt item, and 3 super-categories related to not paying off the item. We found that when developers chose to pay off a code debt item, they decided to do so soon. When they chose not to pay it was a project-specific decision. Also, when developers used the same criterion, the payment priority chosen was in the same neighborhood. Finally, we noted that each project needs a specific set of criteria to prioritize technical debt.
- In Chapter 6, we present a quantitative study that aims to develop machine learning methods to prioritize technical debt items. That is, to decide whether and when a technical debt item should be paid off. We trained nine well-known machine learning methods over 27 features and assessed their performance using traditional and tuned analysis. We found KNN and RF had the best performance for all approaches. DT and XGB are also in the best performance cluster for most of the approaches tested.
- In Chapter 7, we represent the final considerations, contributions, and limitations. We also presents future research suggestions.

Chapter 2

Background

In this chapter, we present the main concepts related to technical debt. First of all, we define and explain the technical debt metaphor. Then, we show ways to classify technical debt using several criteria, such as intention, prudence, short-term, and long-term. Finally, we also offer technical debt properties.

In addition, we present the technical debt management process. This process consists of identifying technical debt and storing the items in a list; measuring each item; monitoring it; making decisions. Lastly, we present the major concepts related to technical debt prioritization.

2.1 The Technical Debt Metaphor

In 1992, Ward Cunningham introduced the technical debt metaphor, comparing it with financial debt. In his narrative (Cunnigham, 1992), Cunningham claims that when the code quality is compromised, it is the same as incurring a debt. He also said that the debt is paid when the code is rewritten. In addition, he makes an alert saying that every minute that debt is not paid, interest is being added to it.

Below is a part of his report:

"Shipping first-time code is like going into debt. A little debt speeds development so long as it is paid back promptly with refactoring. The danger occurs when the debt is not repaid. Every minute spent on code that is not quite right for the programming task of the moment counts as interest on that debt. Entire engineering organizations can be brought to a stand-still under the debt load of an unfactored implementation, object-oriented or otherwise." (Cunnigham, 1992).

In Measuring and Monitoring Technical Debt (Seaman and Yuepo Guo, 2011), Seaman and Guo defined technical debt as a metaphor for an incomplete, immature, or inadequate artifact in the software development lifecycle. We mainly use this definition in this paper.

On April 18-22, 2016, technical debt researchers and specialists met at the Managing Technical Debt seminar in Dagstuhl. After much discussion, they developed the following technical definition:

"In software-intensive systems, technical debt is a design or implementation construct that is expedient in the short term, but sets up a technical context that can make a future change more costly or impossible. Technical debt is a contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability." (*Managing Technical Debt* 2016)

Acquiring a technical debt item could bring short-term benefits (Yuepu Guo, Rodrigo Oliveira Spínola, *et al.*, 2016), such as reducing time, resources, and effort in task development. On the other hand, in the long term, technical debt could cause negative impacts on software quality. Technical debt risks the project, causing higher costs to develop new features and maintain existing code.

The software development teams need to balance the benefits of acquiring technical debt and the cost to pay it off and its interest (Seaman and Yuepo Guo, 2011). The uncertainties associated with technical debt make the decision-making process even more complex. Therefore, technical debt management (identification, measuring, and monitoring) could provide relevant information to the teams to decide whether and when a technical debt item should be paid off. This results in greater visibility for the project, helping to improve the software quality and team productivity to implement new features and perform maintenance effectively.

Technical debt is a metaphor for financial debt, including the vocabulary used to express it. It allows different stakeholders involved in the software project (Ernst *et al.*, 2015) to understand and follow the issues related to technical debt. For this reason, it is easier to show the importance of keeping technical debt under control during project development.

Technical debt only happens in the software being developed or already in production (Yuepu Guo, Rodrigo Oliveira Spínola, *et al.*, 2016). It is impossible to acquire a technical debt of something that does not exist, or it would not be possible to pay it. Thus, abandoned software projects do not accumulate more technical debts. Sometimes, the technical debt acquired during a project is simply canceled. On the other hand, the technical debt may be linked not only to a project. That is, it may also spread to several related projects or directly affect the institutions' capital responsible for its development. Therefore, even if the project fails, the institutions may have to pay at least part of the debt and interest related to the project.

2.2 Technical Debt Classification

In this section, we present the proposed classifications for technical debt. Once the technical debt acquired is known, McConnell (McConnell, 2007) classifies it into short-term technical debt and long-term, depending on the time that the payment is planned.

As financial debt, technical debt can be repaid in **short-term** (Brown *et al.*, 2010). This type of technical debt is often paid to attend to immediate needs or when the team has extra budget or resources.

Another characteristic of this technical debt type is the short frequency in which the payments are made to rehabilitate credit for acquiring new technical debt items. In general, short-term debt does not require significant planning. It is acquired to attend to immediate needs and does not produce a significant negative impact or a considerable accumulation of interest.

In general, **long-term** technical debt items (Brown *et al.*, 2010) are strategically acquired, with planning to attend to high-impact needs. In this case, teams can incur debt for implementing new features, coverage of tests, system failures, lack of adequate technical support, or any other technical need that requires a great team effort. However, these technical debt items will need to be paid off in the future. Even the technical debt payment is planned in the long-term; if the team members have a need, technical debt can be paid off before or handled for years.

According to Martin Fowler (Fowler, 2009), technical debt can be classified by a quadrant with the following parts: reckless and inadvertent; reckless and deliberate; prudent and inadvertent; prudent and deliberate.

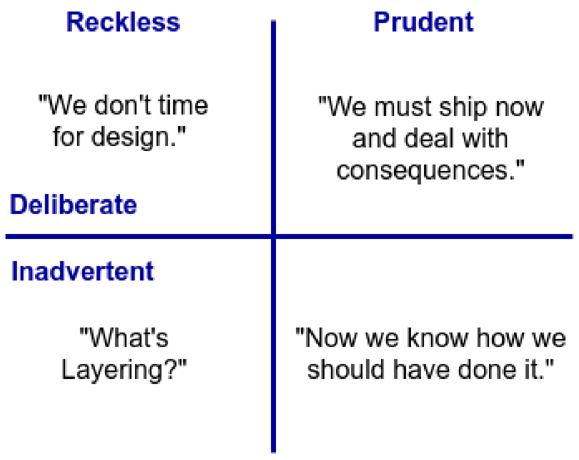

Reckless
Prudent
"We don't time for design."
"We must ship now and deal with consequences."
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it."

Figure 2.1: *Technical Debt Quadrant. (Fowler, 2009)*

Reckless and Inadvertent Technical Debt occurs when the acquired technical debt is unknown and comes from inadvertent actions carried out by the development team without being part of one strategy. It is the most dangerous type of technical debt because the team does not know about the technical debt items or who added them there. That is, it can lead to a bankruptcy project without the team taking notice of the causes.

Reckless and Deliberate Technical Debt occurs when the technical debt to be acquired is known and, even knowing about the negative impact, it is ignored without any strategic reason. Due to imprudence, the technical debt is not paid off, and there is no plan to pay it off later.

Prudent and Inadvertent Technical Debt occurs when the technical debt acquired is unknown, but the development team acknowledges that its actions will accumulate debt. However, the team decided to keep these actions for some strategic reasons. In this case, the team will spend some time learning the correct way to do the technical tasks later. They will also spend time identifying the technical debt that was acquired, being this time counted as interest.

Prudent and Deliberate Technical Debt occurs when the technical debt to be acquired is known and, for some important reason, they decided to acquire it intentionally. Most of the time, the reasons are external to the development team context. Therefore, this type of technical debt results from a previously elaborated strategy. In many cases, the need to assume the debt is mainly related to time. That is the need to deliver a set of tasks in a shorter time than necessary to do them with the desired quality.

2.3 Technical Debt Properties

Principal is the cost to pay off a technical debt item (Yuepu Guo, Rodrigo Oliveira Spínola, *et al.*, 2016). This cost could be calculated in work hours needed to rewrite the code, monetary value, or points. For example, a developer may need to work 4 hours to reduce the complexity of class A.

Interest is the cost of extra work to be done related to a technical debt item (Curtis *et al.*, 2012). Typically, the interest is calculated with the units of measurement as the principal debt. Unlike financial debt, the technical debt interest only is charged when some extra effort is made, depending on the elapsed time.

Probability of interest is the probability of interest being charged (Yuepu Guo, Rodrigo Oliveira Spínola, *et al.*, 2016). Due to the uncertainty related to interest on a technical debt item, there is a need to calculate this probability. For example, a technical debt item has a probability of interest of 25%. That is, on every four commits, the interest cost will be charged.

2.4 Technical Debt Management

The technical debt management process consists in identifying technical debt, measuring it, and making-decision about the payment (Seaman and Yuepo Guo, 2011). This process consists of managing a list of technical debt items.

The technical debt management framework proposed by Carolyn et al. (Seaman and Yuepo Guo, 2011) helps to organize this process. Figure 2.2 shows the framework proposed to identify technical debt items (ID, date, responsible, type, location, and description) and store them in a list. Then, estimating the principal, the interest amount, and the interest probability for each item is necessary. Finally, the list needs to be prioritized to make feasible the decision-making process.

2.4.1 Technical Debt Identification

The first step to managing technical debt is identifying the items. In order to find them, it is possible to use well-defined techniques, methods, and tools. *Identifying and Managing Technical Debt* (Zazworka, Rodrigo O Spínola, *et al.*, 2013) and *Comparing Four Approaches for Technical Debt Identification* (Zazworka, Izurieta, *et al.*, 2014) proposed

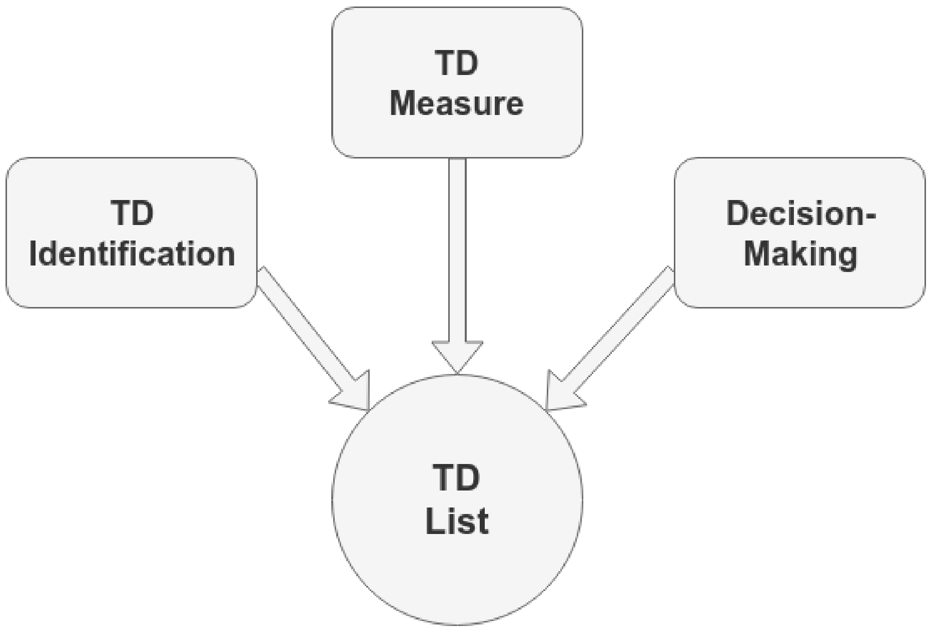


Figure 2.2: *Framework to manage technical debt.* (Seaman and Yuepo Guo, 2011)

technical debt daily indicators and used tools to identify the technical debt items in the source code.

These tools aim to automatically find defects and the absence of patterns in the source code. These elements could generate technical debt or be false positives. The identification tools use the following methods to find the defects: static analysis, code smells, design patterns, modularity violations, and test coverage.

Technical Debt Daily Indicators

The technical debt daily indicators are situations that often happens in the daily development team. The situations are pieces of evidence that there is technical debt in the software. These situations could determine specific technical debt, and they could be easily found in general. However, they also could determine generic technical debt, where it may take a few hours or even days before it can be identified.

Daily identifiers examples are taken from (Zazworka and Seaman, 2013):

- Do not worry about the documentation now;
- The only one that can change this code is John;
- It is right now, but we will need to refactor it later;
- *ToDo* e *FixMe* in the source code;
- We just copy and paste this part;
- Who knows where we stored the database password?;
- I know, if I touch this code it will break;

- We will finish the tests in the next release.

Code Static Analysis

Code static analysis method search for code problems. The analyzed code could be the source or binary (Zazworka, Izurieta, *et al.*, 2014). This analysis is based on violations of best programming practices or design patterns.

By employing static analysis, it is possible to find code snippets that could cause failures. It could also put some software quality aspects down, mainly source code maintenance. Follow bellow some technical debt characteristics examples where the violations found for this method could be associated:

- Bad programming practices;
- Correctness;
- Experimentation;
- Internationalization;
- Malicious code vulnerability;
- Performance;
- Security;
- Multi-thread correctness and
- Style.

Even if the tools find several types of these technical debt items, it is essential to analyze the software context to decide which is important. Thus, we must exclude false positives.

The technical debt found with this method could be related to code defects. However, it also could be related to best pattern violations. This fact directly affects software maintenance. For this reason, the technical debt found with this method could be very interesting to pay off because if the software needs to receive maintenance or new features implementations, there is a high probability of interest being charged. Thus, programmers need more time and effort to correct bugs or implement new features.

Code Smells

The concept of code smells, coined by Martin Fowler and Kent Beck (Fowler, 1999), refers to indications of deeper problems within the affected code area. Most of the time, code smells describe problems related to object-oriented programming and other joint problems involving sets of code lines.

As it involves a variety of problems, several techniques must be used to achieve a method that can identify them manually or automatically. Some problems are defined below:

- Duplicated code.

- Long methods/functions.
- Long classes/modules.
- Too many parameters.
- Data clump.
- Excessively long identifiers.
- Cyclomatic complexity.

In general, the problems found demand refactoring to be fixed. Therefore, the technical debt found with this method is strongly related to maintenance, readability, and reuse. Besides that, many criteria used also related to reliability, security, and efficiency.

This method only allows finding some of the technical debt items automatically. However, it is possible to find them with computer assistance. It occurs because the techniques could identify some false positives or be a technical debt only in a specific context.

Design Patterns

The design patterns are intended to make the source code easier to read and understand. Thus, maintenance is more manageable, and the code is less prone to defects and failures (Zazworka and Seaman, 2013). The method applies standardization and description of how classes can work together.

The design patterns violation can be considered as architectural technical debt. Design pattern concepts are often related to high-level software abstraction involving modules, classes, and methods. Thus, several times, it is necessary to do a complex analysis to identify violations in the pattern. Therefore, it is hard to find tools to identify design patterns violation automatically. However, it is possible to use the design pattern concept and application to assist in the identification. Guides like (Hunt and Hunt, 2013) can help you find patterns like:

- Abstract factory;
- Prototype;
- Adapter;
- Facade;
- Strategy;
- Scheduler.

Tests

The test method shows code snippets that do not have the expected behavior in some situations. Therefore, if a test fails, it can indicate a defect in the software. These defects could be both errors and property violations. In critical software, it is necessary to test using formal verification to ensure that the code satisfies the model. On the other hand, less complex software could be tested using only validation (Delamaro *et al.*, 2007).

Tests could identify technical debt because they could be written to identify quality properties validation. Therefore, a test could identify if a code snippet does not behave as expected.

Besides that, the tests can identify problems in a high-level abstraction called architectural technical debt. For instance, a test could determine if a module complies with the software architecture or if a class behaves according to the expected.

The test's absence, low coverage level, incompleteness, or inadequacy may be considered technical debt. Writing and maintaining tests incur development costs and their absence or inadequacy could impact software quality writing and maintenance of tests' principal technical debt calculation.

Other Methods

According to the software context, another identification method can be necessary to find technical debt items. It could happen because the context may cause a technical debt type to arise or even new types of technical debt to appear.

2.4.2 Technical Debt Measure

In order to measure the technical debt, one possibility is to apply the method described in *The SQALE Methods* (Letouzey, 2012b). This method consists of rules to make the measure feasible and standardize it. Besides that, it standardizes controls related to source code and value aggregation rules.

For each rule, it needs to define a function to compute the cost to pay off a technical debt item. This function is called the remediation function. Computing each item's technical debt principal cost is possible by applying the remediation function. With these results, it is possible to compute either the entire technical debt principal cost of the software or group them according to criteria and obtain the technical debt of part of the software.

Remediation Function

The purpose of the remediation function is to determine the cost of moving from the current state to the desired state of quality (Letouzey, 2012b). Therefore, each quality property or rule is associated with a remediation function. This function can compute the technical debt principal effectively and in a standard way.

The remediation function parameter is a list with an indicator of whether the quality property has been violated and the values associated with it if there are. The function returns a cost value. This cost could be monetary, work-hours cost, or symbolic cost, according to the need of each project.

The remediation function could be a simple constant function that returns either 0 if no violation occurs or a constant value otherwise. Another possibility is to define a multiplicative factor to the violation according to the associated values parameters.

The remediation cost function can also define a more complex function, such as exponential, logarithmic, trigonometric, or any other function. The function must adequately model the cost of paying the technical debt. The more complex functions usage could help in this modeling. For example, the software complexity could increase the payment cost exponentially.

For instance, suppose the quality property is: "Every method must be covered by at least one unit test". It is possible to define the following remediation cost: $f(x, y) = xln(y)$, where either x is 0 if the method was covered or 1 if the method was not covered. The y is the number of lines of the method. The function $f(x, y)$ returns the hour-cost to write a test to cover the method.

It is important to note that any remediation function must be 0 if the quality property was not violated.

Technical Debt Item Storage

In order to aim to organize and compute the technical debt automatically is necessary to use a data structure similar to the one proposed in *Measuring and Monitoring Technical Debt* (Seaman and Yuepo Guo, 2011). The data structure could be modified for each model or context to suit the needs.

ID	42
Date - Time	01/06/13 11:30:34
Quality Property	All method must be covered by at least one test.
Place	Class ABC : Method XYZ
Principal	2 hours e 30 minutes
Interest	30 minutes
Interest Probability	Medium
Payment Priority	High

Table 2.1: *Technical debt record example*

Technical Debt Principal Calculation

According to the SQALE Method (Letouzey, 2012a), in order to compute the entire technical debt principal precisely, it is necessary the following:

- All quality type is independent. That is, a technical debt item cannot be associated with two types of quality;
- All the characteristics of the model must be independent;
- All sub-characteristics of the model must be independent;
- All model quality properties must be independent;
- From the quality properties, all technical debt items must be found.

In practice, the above rules are challenging to enforce. Thus, to perform the calculation, we could compute part of the technical debt of a given quality property, sub-characteristic, characteristic, or type of quality that meets the requirements.

If it is not possible to guarantee independence and that all technical debt is found, we will have:

- Upper boundary: This scenario assumes that all technical debt can be found, but independence cannot be guaranteed. It is due to the intersection between technical debt items, causing costs to be added more than once;
- Lower boundary: In this scenario, independence can be guaranteed, but it cannot be guaranteed that all debt has been found. In this case, the technical debt will be counted only once and may not be less than the calculated value. On the other hand, as some items may not have been found, the technical debt may be higher than the calculated value;
- Approximation: This scenario arises when independence can not be guaranteed and it is not possible to find all technical debt items. It is not the ideal case but the most common one. In this case, it can not be a higher boundary since technical debt items were not found could exceed this limit. On the other hand, it can not be a lower boundary, as there may be intersections between technical debt items, so the value found approximates the actual value of technical debt.

The technical debt calculation can be done on different levels, from the lowest quality properties to the highest, calculating the debt of the software as a whole. The item could be grouped into several levels to create technical debt indexes. These indexes standardize the sum of several measures into a set that makes sense and is essential for certain parts of interest. For example, the debt can be grouped for each module or file; or characteristics or sub-characteristics.

Step-by-step to Calculate Technical Debt

The technical debt principal (Letouzey, 2012a) can be computed by the following formula:

$$DT = \sum_{i=1}^{n} f_i$$

where f_i is the cost to do a property i to be valid in every items; computed by the following way:

$$f_i(P_1, P_2, ..., P_{m_i}) = \sum_{j=1}^{m_i} h_i(P_j)$$

where P_j is a vector that describes each evaluated technical debt item and

$$h_i = \begin{cases} 0, & \text{if i was not violated,} \\ x > 0, & \text{otherwise.} \end{cases}$$

represents the cost of making the valuation item (P_j) satisfy property i.

Group Technical Debt Items

It is often interesting to know the technical debt for each system characteristic. Group the technical debt into sub-characteristics directly associated with a characteristic.

In **The SQALE Methods** (Letouzey, 2012a), an index is created for each characteristic representing the related properties' technical debt sum. The sub-characteristics properties are also included.

The followings characteristics created these indexes:

- Testability (STI);
- Reliability (SRI);
- Changeability (SCI);
- Eficiency (SEI);
- Security (SSI);
- Maintenence (SMI);
- Portability (SPI);
- Reuse (SRuI).

In a software project, we can have several stakeholders. Thus, it could need to group indexes to reflect something of specific interest to any of these stakeholders.

For instance, a stakeholder might be interested in knowing if the system is working correctly. Therefore, he might want to see the grouping of technical debt of testability and reliability together, thus generating a new index:

$$SCRI = STI + SRI.$$

Technical Debt Dimension

It is possible to create density indexes to dimension the technical debt against the source code. These indexes are defined as the division of the absolute indexes by the possible total measure for each of the properties associated with the index.

For example, given the absolute testability index $STI = 50$, paying all technical debt items related to testability would cost 50 minutes. Therefore, it is necessary to find the total hours that would be spent if no test had been performed. Suppose that this value is 500 minutes. Thus, the testability density index $STID = \frac{50}{500} = 0.1$, the technical debt of testability is present in 10% of the code.

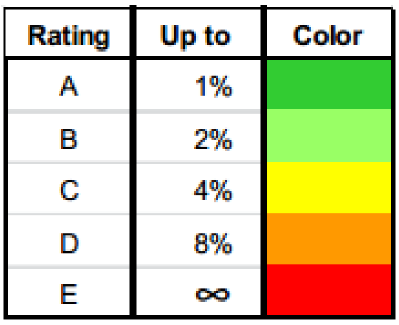
This technique could be applied to all absolute indexes. It is possible to apply in a grouping of indexes. For this reason, it is possible to create a density index for measuring technical debt.

In practice, small codes may have high-density indices. Large codes may have low-density indices even having a large number of technical debt items.

Technical Debt Graphical Representations

The technical debt principal can be represented at several scales or using artifacts such as graphs and pyramids.

SQALE Rating SQALE (Letouzey, 2012a) uses three rate unities: classification, percentage, and color. The rate is typically divided into five or more values, as shown in Figure 2.3. This rate is directly applied to the density indexes and quickly verifies the degree of technical debt at points of interest.



Rating	Up to	Color
A	1%	🟩
B	2%	🟩
C	4%	🟨
D	8%	🟧
E	∞	🟥

Figure 2.3: *Table of grades and colors of the SQALE rate. (Letouzey, 2012a)*

For example: Suppose the testability index (STI) is equal to 18 and that all possible points add up to 600. Then, the density index STID is equal to 0.03, which in percentage equals 3%. Thus, its classification is C and its color is yellow.

Kiviat Graph (Letouzey, 2012a) Suppose the testability index (STI) is equal to 18 and all possible points add up to 600. Then, the density index STID equals 0.03, which in percentage equals 3%. Thus, its classification is C, and its color is yellow.

To assemble this graph, follow these steps:

1. For each characteristic, mark the point where the technical debt is;
2. Link all points of the technical debt, thus obtaining a polygon of the current technical debt;
3. For each characteristic, mark the point with the maximum wanted the value to the technical debt and;
4. Link all points of maximum values, obtaining a polygon of the maximum value of technical debt.

If the polygon of the current technical debt is entirely included within the polygon of the maximum technical debt means that the technical debt is under control. If part of the technical debt polygon is out, it means that the characteristic related to the point that is out has passed the acceptable levels and needs to be paid off.

SQALE Pyramid (Letouzey, 2012a) The characteristics can be placed in importance order. Then it is possible to construct a pyramid with the following characteristics:

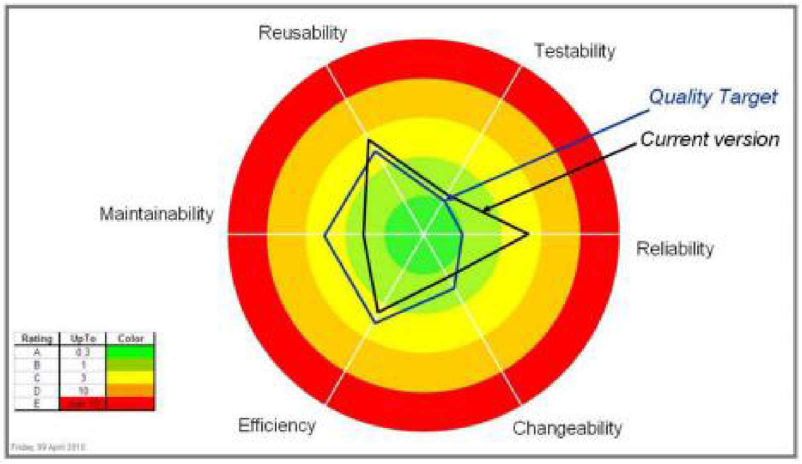


Figure 2.4: *Kiviat Graph for representation of technical debt by characteristics proposed by the SQALE method. (Letouzey, 2012a)*

- Each field of each line receives the technical debt value of the associated characteristic.
- All the debt in the column must be added together in the last line.

This pyramid allows visualizing the accumulated debt of different characteristics, as shown in Figure 2.5.

2.4.3 Technical Debt Monitor

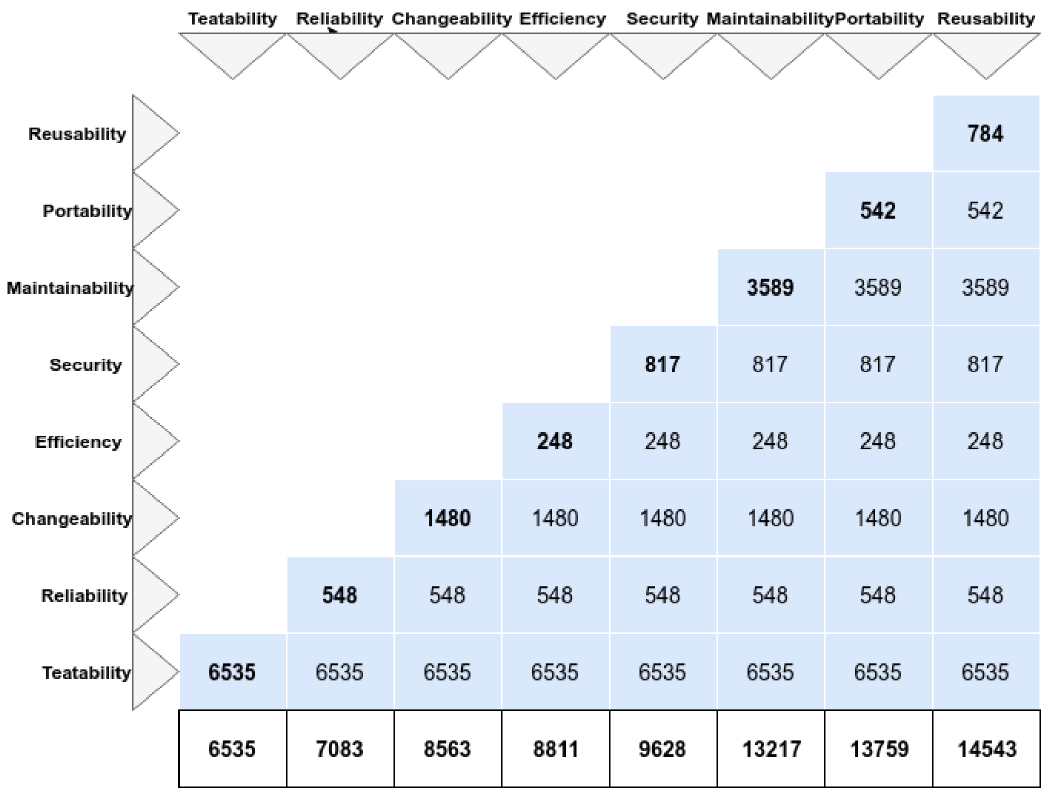
Technical debt monitoring could be done in two ways. The first is immediate, and the second is over time. The monitor also could be done in an explicit technical debt in the item list or to monitor to find implicit technical debt (A. Freire, n.d.). What should be monitored may vary according to the stakeholders. For example, the system buyer may only be concerned with the total technical debt of the current project. On the other hand, the responsible for testing the system may want only to see the technical debt related to testing and its evolution over time. The technical debt indexes could help in the monitoring. With them, it is possible to analyze specific software parts quickly. Thus each stakeholder could analyze the technical debt from their point of view.

Immediate Monitoring

After identifying the technical debt items and processing them using the remediation functions to compute the principal cost, the result is a technical debt list.

Technical debt immediate monitoring consists of analyzing the technical debt to make decisions. This monitoring allows for verifying the points with more significant technical debt problems. That is, where there is a considerable amount of technical debt with the urgency of payment.

In the immediate monitoring, the indexes are essential to make-decision. They allow easy verifying of the amount of technical debt at a given point.



	Teatability	Reliability	Changeability	Efficiency	Security	Maintainability	Portability	Reusability
Reusability								**784**
Portability							**542**	542
Maintainability						**3589**	3589	3589
Security					**817**	817	817	817
Efficiency				**248**	248	248	248	248
Changeability			**1480**	1480	1480	1480	1480	1480
Reliability		**548**	548	548	548	548	548	548
Teatability	**6535**	6535	6535	6535	6535	6535	6535	6535
	6535	**7083**	**8563**	**8811**	**9628**	**13217**	**13759**	**14543**

Figure 2.5: *Technical debt pyramid of the SQALE methodology.* (LETOUZEY, 2012a)

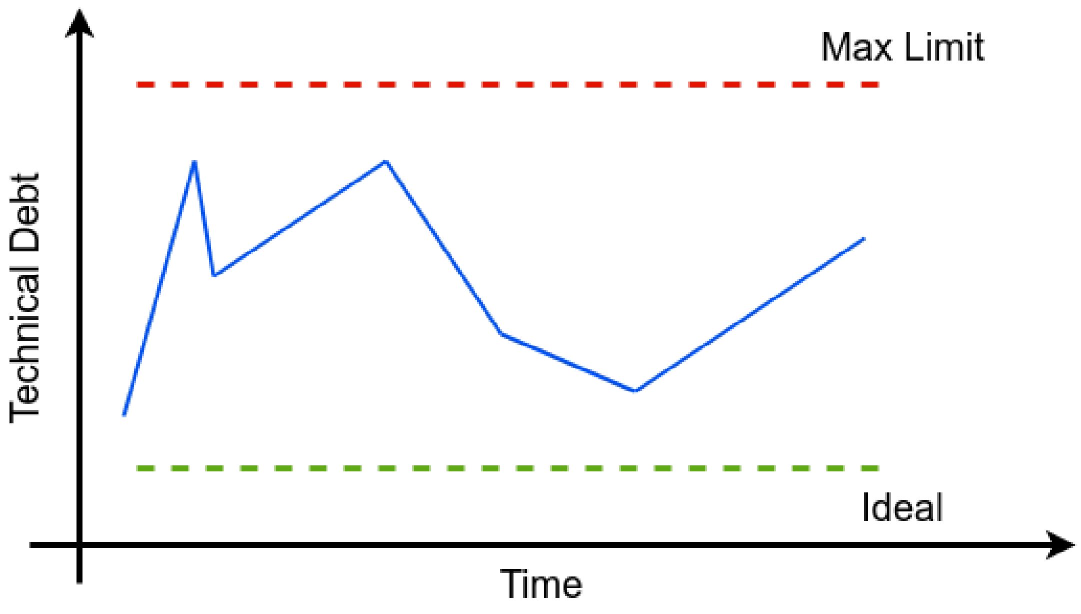


Figure 2.6: *Line chart to monitor technical debt over time.* (KNIBERG, 2013)

Monitoring Over Time

Monitoring the technical debt over time consists of gathering the technical debt lists of different dates to analyze them in different ways. The first goal is to follow the technical debt evolution.

This type of monitoring is essential to have control of the technical debt. Monitoring over time allows us to keep it from increasing at critical points where payment priority is high. This process could prevent the project from going bankrupt.

Indexes are also significant for monitoring over time. With the indexes of different dates, it is possible to verify the evolution of technical debt at key points. Therefore, through the indexes, it is possible to take measures to inhibit or even decrease some specific technical debt growth.

Monitoring Methods Examples

Several methods could be used to monitor technical debt. Some of them are more specific, monitoring direct technical debt items. Other are more subjective monitoring factors that could impact the technical debt. Below, we present a list of monitoring methods:

- ***Tracker*** has carried out specific monitoring for each technical debt item. For each item, it is necessary to check some points such as the priority to pay the technical debt; when the item will be paid off; by whom it will be paid; and additional information. The tracker helps to plan the technical debt payment and provide a payment history.
- ***Kanban*** is a tool to track the tasks of a project. With the technical debt identified, it is possible to place each item, or a group of items, as a task to be performed. Therefore, monitoring which technical debts have been, are being, or will be paid off can be done simply using*kanban*.
- **Customer Happiness Monitor** is a subjective method and does not require identifying technical debt. However, customer happiness could be a clue as to whether the project is going in the right direction and, consequently, without excess technical debt. When a client begins to get impatient, angry, or disinterested in the project, it may indicate that the project has too much technical debt that directly affects the client's mood.
- **Changing Difficult Monitor**, when the development team begins having difficulty changing the project, maintaining or implementing new features, indicates that technical debt may be high or increasing. This monitoring mainly concerns the technical debt of the characteristics of reuse, maintenance, and changeability. For this method, it is necessary to check if the current developers' capabilities are equivalent to previous developers. If they are not equivalent, a technical improvement may be needed.
- **Team Monitors**, the unhappy team, working overtime, disorganized, having technical problems, or any other problem may have their projects hampered. In particular, the source code may be affected by technical debt due to fatigue, inattention, technical inability, or other factors related to the team members.

- **Retrospectives**, conducting retrospectives with those involved in the project is essential to identify problems and verify that everyone's expectations have been met. The problems exposed in these events should be investigated to verify their impact on the quality of the product. These problems are indications of technical debt.
- **Commits over Time** is a method for verifying whether the team is committed to the project, checking the complexity of implementing new features or performing maintenance, and tracking the quantity and quality of the commits over time. If the curve decreases, it may indicate the project has technical debt since writing a line of code may have become more costly.

It is important to note that many of these methods are intuitive and require human analysis to identify whether the detected problems are related to technical debt. For example, the number of commits could drop sharply from month to month as demand for new features may have declined.

2.5 Technical Debt Prioritization

Prioritization is a crucial aspect of technical debt management as it helps decide whether and when a technical debt item should be paid off, resulting in an ordered list of technical debt items (Yuepu Guo, Seaman, *et al.*, 2011). The items that should be paid first must be on top of the list. Some technical debt prioritization methods could define a threshold, dividing the list into items that should be paid and the items that not should be paid. Most of this decision focus on maintenance tasks that have an immediate impact on the production software, thus it has financial visibility (Seaman, Yuepu Guo, *et al.*, 2012).

Efficient technical debt management requires technical debt prioritization since it is not feasible to pay off all items (Charalampidou *et al.*, 2017). According to the context of the project, some items need to be paid earlier than others, because they have a higher probability of accruing interest. Besides that, some technical debt items do not even need to be paid, because they have a low probability of accruing interest.

2.6 Chapter Summary

This chapter presents the main concepts, classifications, and properties related to technical debt. We also presented the technical debt management process. Lastly, we presented the significant concepts related to technical debt prioritization.

First of all, this chapter introduced the technical debt metaphor and concepts. We presented the first definition formulated by Ward Cunningham. We also presented Seaman et al. and Dagstuhl seminar technical debt definition. Second, we presented the classification of technical debt and its properties.

We also presented methods to identify, store, measure, and monitor technical debt. These methods make up the technical debt management process. Lastly, we presented the main concepts related to technical debt prioritization.

The next chapter will present a mapping study on prioritizing technical debt.

Chapter 3

Systematic Mapping Review on Technical Debt Prioritization

Several methods for identifying, measuring, and monitoring technical debt have been developed, and new tools have been created to automate the process in the last few years. Thereby, it is possible to find thousands of technical debt items in each software project that could take hundreds of days to be paid off (Curtis *et al.*, 2012).

In order to find previous work, we performed a mapping study to find discussions and methods that aim to help prioritize technical debt. In other words, methods to identify which technical debt items should be paid and when the payment should be made and to supply information to the decision-making process. We analyzed the main papers of the area, providing a classification of them. The main result is to highlight the current prioritization methods and to compare them, allowing the improvement of existing methods.

We based this chapter on (Pina, Goldman, and Tonin, 2021). We updated the search for the systematic mapping review to May 2023.

3.1 Research Methodology

We followed the steps proposed by Petersen et al. (Petersen *et al.*, 2008) to perform a systematic mapping in the current literature. We performed all six steps: definition of the research question; conduct search; screening papers; data extraction and mapping process; classification schema; and systematic map. Figure 3.1 shows Petersen's process.

3.1.1 Conduct Search

We used the query ("technical debt") AND (prioritization OR decision-making) to o search for papers in the principal computer science papers' bases: ACM Digital Library (ACM DL), IEEE Xplore, Science Direct, Scopus, Springer Link, and Web of Science. In IEEE Xplore, we broke down the query into two: (“technical debt") AND (prioritization); (“technical debt") AND (decision-making).

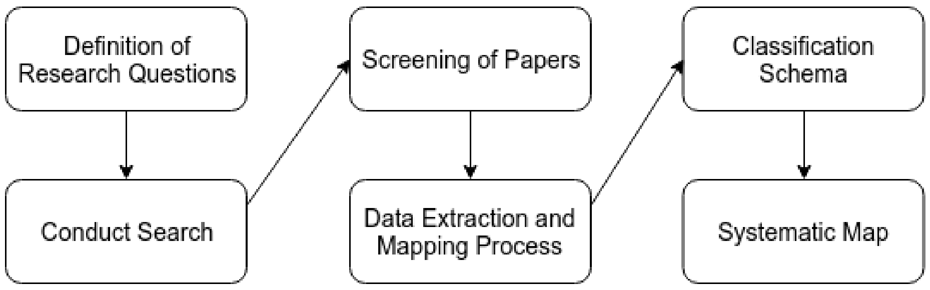


Figure 3.1: *Petersen's process to perform a systematic map. (Petersen et al., 2008)*

The process of building the search query was iterative. First, we searched only considering the term ("technical debt"), and several papers were listed, but most were not related to technical debt prioritization. Then, we changed the query to ("technical debt") AND (prioritization) to filter only the papers related to prioritization. Analyzing a sample, we identified that the term "decision-making" is sometimes synonymous with prioritization. Then, we added the expression to the query as an alternative to prioritization. The application of all these processes aids in the completeness of the search.

We searched for them using the default fields: "any field" in ACM DL "metadata only" in IEEE Xplore; "keywords" in Science Direct; "article title, abstract, keywords" in Scopus; "with all of the words" in Springer Link and "topic"; in Web of Science.

3.1.2 Screening of Papers

The screening process consisted of two steps. First, we established the minimum criteria to accept a paper. Then, we established criteria to exclude the ones that were not relevant.

The inclusion criteria used were:

- The abstract must explicitly refer to at least one prioritization method or decision criteria regarding the technical debt payment;
- The paper must be written in English;
- The paper must have been published in a journal, conference, or workshop.

The exclusion criterion used was:

- Papers with more than ten years. Before that, the term "technical debt" was not widely used.

3.1.3 Data Extraction and Mapping Process

The total number of papers found was 1027. After applying the inclusion and exclusion criteria, we selected 146 papers. Then, we removed the duplicated papers, resulting in **70 unique papers**. Table 3.1 shows a search results overview. Figure 3.2 shows the distribution of selected studies over time.

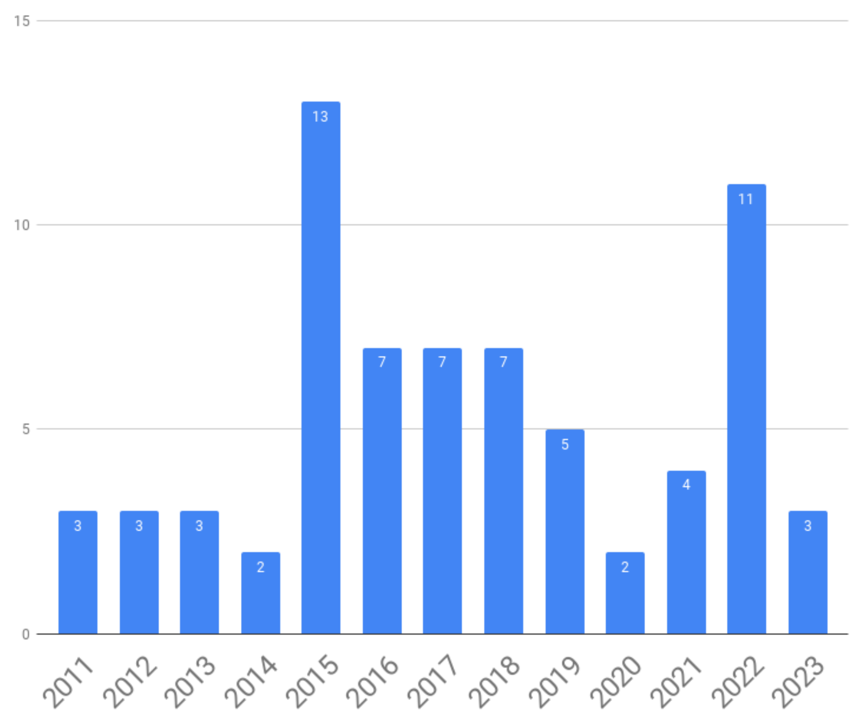


Figure 3.2: *Distribution of selected studies over time.*

Base	**Total**	**Total Selected**
ACM DL	225	21
IEEE Xplore	97	41
Science Direct	271	8
Scopus	140	43
Springer Link	245	5
Web of Science	65	28
Total	**462**	**146**
Total of Unique Papers		**70**

Table 3.1: *Search results overview*

3.1.4 Classification Schema

We applied the keywording process described by Peterson et al. (Petersen *et al.*, 2008). We used the full text of the selected papers instead of only the abstract. We clustered and linked the keywords to develop a technical debt prioritization taxonomy. Afterward, we identified some generic codes and classified them in the first level. We also identified more specific others and classified them into a second level. Also, we related the first level to the second level. From the clusters, we proposed categories to classify the papers.

3.2 Mapping Review Results

This section presents the results found in the mapping review and the answers to the following research questions.

3.2.1 RQ1: What methods and techniques were proposed to prioritize technical debt?

We have grouped methods and techniques to prioritize technical debt from the selected papers, developing a two-level abstraction with linked categories.

The taxonomy, presented in Figure 3.3, is divided into two levels. The first level is the immediate classification of the papers. The second level is the specific sub-classification derived from the secondary aspects of the paper approaches.

The categories of the second level are linked to at least one category in the first level. A continuous line means that all papers are related to that category. A dashed line means that only a few papers are related to that category.

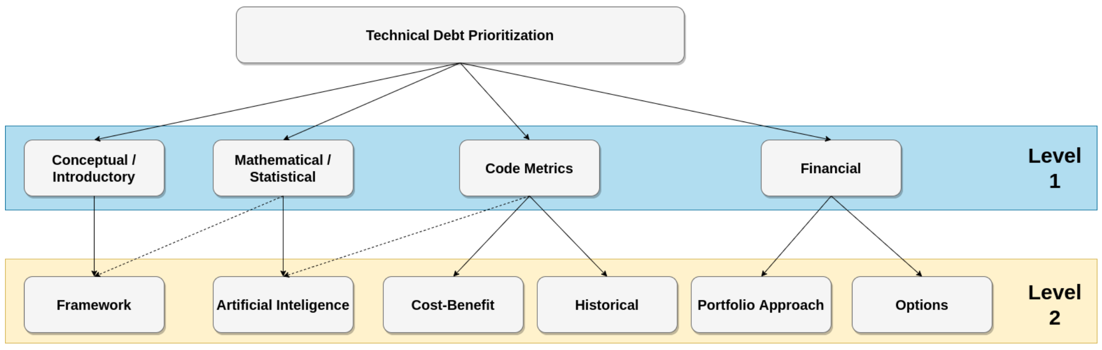


Figure 3.3: *Technical debt prioritization taxonomy*

Each paper is in the best-fit category in the first level. According to their specific aspects, some papers can also be in one category of the second level.

3.2.2 First Level

We have four categories in the first level: Conceptual / Introductory, Mathematical/Statistical, Code Metrics, and Financial. Table 3.2 shows the papers mapped into the first-level taxonomy.

Classification	**Papers**
Conceptual /Introductory	(Gomes *et al.*, 2011), (Seaman, Yuepu Guo, *et al.*, 2012), (Morgenthaler *et al.*, 2012), (Falessi, Shaw, *et al.*, 2013), (Daneva *et al.*, 2013), (Ernst *et al.*, 2015), (Martini and Bosch, 2015a), (Martini and Bosch, 2015b), (Leppanen *et al.*, 2015), (Fernández-Sánchez, Garbajosa, *et al.*, 2015), (Martini, Bosch, and Chaudron, 2015), (Riegel and Doerr, 2015), (Martini and Bosch, 2016), (Garousi and Mäntylä, 2016), (Brauer *et al.*, 2017), (Martini and Bosch, 2017), (Hormann *et al.*, 2017), (Becker *et al.*, 2018), (R. d. Almeida *et al.*, 2018), (Pina, Seaman, *et al.*, 2022), (S. Freire, Rios, Pérez, Torres, *et al.*, 2021), (Mandic *et al.*, 2021), (M. Stochel *et al.*, 2022), (Albuquerque *et al.*, 2022), (Wiese *et al.*, 2022), (Pérez *et al.*, 2021), (S. Freire, Rios, Pérez, Castellanos, *et al.*, 2023), (Alfayez, Winn, *et al.*, 202 (Costa *et al.*, 2022), (Tsintzira *et al.*, 2020), (De Toledo *et al.*, 2022)
Mathematical /Statistical	(Schmid, 2013), (Fontana *et al.*, 2015), (Skourletopoulos, Chatzimisios, *et al.*, 2015), (Akbarinasaji, 2015), (Mohan *et al.*, 2016), (Codabux and Williams, 2016)
Code Metrics	(Zazworka, Seaman, and Shull, 2011), (Snipes *et al.*, 2012), (Falessi and Voegele, 2015), (Chatzigeorgiou *et al.*, 2015), (Skourletopoulos, Mavromoustakis, *et al.*, 2016), (Yli-Huumo *et al.*, 2016), (Choudhary and P. Singh, 2016), (Charalampidou *et al.*, 2017), (L. F. Ribeiro *et al.*, 2017), (Haendler *et al.*, 2017), (Mensah *et al.*, 2018), (Plösch *et al.*, 2018), (Detofeno *et al.*, 2022)
Financial	(Yuepu Guo and Seaman, 2011), (Fernández-Sánchez, Díaz, *et al.*, 2014), (Abad and Ruhe, 2015), (Brauer *et al.*, 2017)

Table 3.2: *Papers mapping into the first level taxonomy*

Papers in the **Conceptual/Introductory** category discuss the concepts involved in prioritizing technical debt without specifying an applicable method, both from the point of view of development teams and management.

Several theoretical surveys and discussions based on interviews, questionnaires, and observations were carried out to identify the elements related to the prioritization and decision-making process (Ernst *et al.*, 2015; Martini and Bosch, 2015b; Leppanen *et al.*, 2015; S. Freire, Rios, Gutierrez, *et al.*, 2020; Pina, Seaman, *et al.*, 2022; S. Freire, Rios, Pérez, Torres, *et al.*, 2021; S. Freire, Rios, Pérez, Castellanos, *et al.*, 2023; De Toledo *et al.*, 2022). On the other hand, the papers (Riegel and Doerr, 2015; Garousi and Mäntylä, 2016; Becker *et al.*, 2018; Costa *et al.*, 2022; Tsintzira *et al.*, 2020) did an SLR to synthesize some aspects of management and prioritization. Alfayez et al. (Alfayez, Winn, *et al.*, 2023) studied how SonarQube prioritizes technical debt.

Some papers are mainly conceptual. They define formal ways to manage technical debt and to perform prioritization. (Gomes *et al.*, 2011; Morgenthaler *et al.*, 2012; Martini and Bosch, 2015a). On the other hand, Falessi et al. (Falessi, Shaw, *et al.*, 2013) and Stochel et al. (M. Stochel *et al.*, 2022) present a practical way to carry out this process without a specific method or tool.

Seaman et al. (Seaman, Yuepu Guo, *et al.*, 2012) introduced a possible method to prioritize technical debt. Other papers try to define a step-by-step method for selecting technical debt items to be paid off (Leppanen *et al.*, 2015; Martini, Bosch, and Chaudron, 2015; Fernández-Sánchez, Garbajosa, *et al.*, 2015; Martini and Bosch, 2016; Brauer *et al.*, 2017; Martini and Bosch, 2017; Hormann *et al.*, 2017; R. d. Almeida *et al.*, 2018; R. R. d. Almeida *et al.*, 2019; R. d. Almeida, 2019; M. G. Stochel, Cholda, *et al.*, 2020; Wiese *et al.*, 2022), defining a framework for it.

There is only space for papers in the conceptual/introductory category that present new ways of prioritizing technical debt conceptually.

Papers in the **Mathematical/Statistical** category use mathematical and statistical methods to prioritize technical debt. Most papers in this category apply already-known methods and models used in other areas. Other papers created a new formalization for technical debt to make the decision-making process feasible (Schmid, 2013).

Some methods are based on well-known techniques, such as arithmetic formulas (Skourletopoulos, Chatzimisios, *et al.*, 2015) and quantile analysis (Fontana *et al.*, 2015). On the other hand, some methods use artificial intelligence (Mohan *et al.*, 2016), and machine learning (Akbarinasaji, 2015; Codabux and Williams, 2016) to predict the technical debt payment.

Papers in the **Code Metrics** category use code metrics and historical analysis to prioritize technical debt. Code metrics are widely used to perform analyses and to make comparisons in software engineering. Therefore, the methodologies for treating code metrics are well-known and diffused for academics and practitioners.

These papers use code metrics to quantify each technical debt item's principal and interest cost. This quantification is used with other techniques, such as cost-benefit ranking (Zazworka, Seaman, and Shull, 2011; Yli-Huumo *et al.*, 2016; Plösch *et al.*, 2018), Change

Control Boards (CCB) (Snipes *et al.*, 2012), and mathematical formulas (L. F. Ribeiro *et al.*, 2017; Skourletopoulos, Mavromoustakis, *et al.*, 2016) to prioritize the technical debt items. Data mining and historical analysis are also widely used techniques to prioritize technical debt (Falessi and Voegele, 2015; Choudhary and P. Singh, 2016; Mensah *et al.*, 2018; Detofeno *et al.*, 2022; Tsoukalas, Siavvas, *et al.*, 2023; B. d. Lima *et al.*, 2022; Katin *et al.*, 2022).

Lastly, papers in the **Financial** category use metaphors to make the relationship between technical and financial debt. These papers take advantage of the knowledge and methods developed within the finance area. They apply methods used in finance and business areas, such as the portfolio approach (Brauer *et al.*, 2017; Yuepu Guo and Seaman, 2011; Plösch *et al.*, 2018), and options (Fernández-Sánchez, Díaz, *et al.*, 2014; Abad and Ruhe, 2015; Aldaeej and Seaman, 2018) to prioritize technical debt items. Some papers (De Almeida *et al.*, 2021; Da Silva *et al.*, 2022; M. G. Stochel, Wawrowski, *et al.*, 2022) used financial terms to define the prioritization process.

3.2.3 Second Level

The second level has six categories: Framework, Artificial Intelligence, Cost-Benefit, Historical, Portfolio Approach, and Options. Table 3.3 shows the papers mapped into the second-level taxonomy.

Classification	**Papers**
Framework	(Leppanen *et al.*, 2015), (Fernández-Sánchez, Garbajosa, *et al.*, 2015), (Martini, Bosch, and Chaudron, 2015), (Martini and Bosch, 2016), (Brauer *et al.*, 2017), (Martini and Bosch, 2017), (Hormann *et al.*, 2017), (R. d. Almeida *et al.*, 2018), (R. R. d. Almeida *et al.*, 2019), (R. d. Almeida, 2019), (M. G. Stochel, Cholda, *et al.*, 2020), (S. Freire, Rios, Gutierrez, *et al.*, 2020), (Mandic *et al.*, 2021), (Wiese *et al.*, 2022)
Artificial Intelligence	(Akbarinasaji, 2015), (Codabux and Williams, 2016), (Mohan *et al.*, 2016), (Kouros *et al.*, 2019), (Alfayez and Boehm, 2019)
Cost-Benefit	(Zazworka, Seaman, and Shull, 2011), (Snipes *et al.*, 2012), (Skourletopoulos, Mavromoustakis, *et al.*, 2016), (Yli-Huumo *et al.*, 2016), (L. F. Ribeiro *et al.*, 2017), (Plösch *et al.*, 2018)
Historical	(Falessi and Voegele, 2015), (Choudhary and P. Singh, 2016), (Charalampidou *et al.*, 2017), (Tornhill, 2018)
Portfolio Approach	(Yuepu Guo and Seaman, 2011),(Plösch *et al.*, 2018) (Brauer *et al.*, 2017), (Albarak and Bahsoon, 2018)
Options	(Fernández-Sánchez, Díaz, *et al.*, 2014), (Abad and Ruhe, 2015), (Aldaeej and Seaman, 2018)

Table 3.3: *Papers mapping into the second level taxonomy*

The papers discuss ways to standardize the technical debt prioritization process in the **Framework** category. Most of the frameworks developed consist in finding technical debt items, measuring their payment cost and interest (Fernández-Sánchez, Garbajosa, *et al.*, 2015) (Leppanen *et al.*, 2015) (Martini and Bosch, 2016). Using their steps, or its preemption, one may prioritize technical debt payment and decide how items must be paid off.

Some papers deal with a temporal decision, for instance, when-to-release decisions (Brauer *et al.*, 2017), architectural debt evolution (Martini, Bosch, and Chaudron, 2015), and technical debt evolution and contagious phenomenon (Martini and Bosch, 2017).

All papers in the Framework category are related to Conceptual/Introductory ones because they also discuss conceptual and introductory aspects to define their framework. Some papers are related to the Mathematical/Statistical categories because they also use mathematical formalization to make the process feasible (Martini and Bosch, 2016; Brauer *et al.*, 2017).

Papers in the **Artificial Intelligence** category use well-defined methods, such as a random search, hill-climbing, and simulated annealing, to find solutions to prioritization (Mohan *et al.*, 2016). Besides that, machine learning algorithms are also used to train predictors to infer technical debt prioritization (Akbarinasaji, 2015) (Codabux and Williams, 2016). Search-based methods were also used to prioritize the technical debt items (Alfayez and Boehm, 2019; Kouros *et al.*, 2019).

The papers in the Artificial Intelligence category are related to the Mathematical/Statistical ones because they use search methods and machine learning to prioritize technical debt. The category is also related to Code Metrics because it uses code measures to provide input to the methods.

Papers in the **Cost-Benefit** category deal mainly with simple cost-benefit analysis. The method compares costs related to maintaining a technical debt item and the benefits of paying it off. Generally, a rank is built with technical debt items where the better cost-benefit items are on top. All papers in this category are related to Code Metrics because they also use code measures in their methods.

Papers in the **Historical** category use historical data to define ranges and parameters. The papers use code changes, metrics values, and evolution to classify the software and identify the technical debt (Falessi and Voegele, 2015) (Charalampidou *et al.*, 2017) (Choudhary and P. Singh, 2016) (Tornhill, 2018) to pay off.

Papers in the **Portfolio Approach** category use financial concepts and handle the technical debt items as a bundle of assets held by an investor. Investors use the portfolio strategy to try to reduce risks.

The method determines which types and the number of assets - in this case, technical debt items - should be invested. All papers in this category (Yuepu Guo and Seaman, 2011; Brauer *et al.*, 2017; Plösch *et al.*, 2018; Albarak and Bahsoon, 2018) are related to the Financial category because they use economic approaches and metaphors to prioritize technical debt.

Papers in the **Options** can be seen as an investment decision process. As technical debt, options take a greater certainty in the short-term and more significant uncertainty in the long term. Therefore, paying off a technical debt item is similar to buying an option to facilitate future changes using available resources.

Fernández-Sánchez et al. (Fernández-Sánchez, Díaz, *et al.*, 2014) created a decision tree to decide if it is worth paying off the architectural technical debt. Abad and Ruhe (Abad and Ruhe, 2015) use a binomial model to apply real options in technical debt management in software engineering requirements. All papers in this category are related to the Financial category because option decision is a method used in finance. Aldaeej and Seaman (Aldaeej and Seaman, 2018) use real options analysis to make a decision.

3.2.4 RQ2: What results do technical debt prioritization methods provide?

We categorized the prioritization methods presented in the selected papers according to the result provided. We found three kinds of results: boolean, category, and ordered list.

- **Boolean**: technical debt items are divided into those that should be paid and those that should not be paid;
- **Categories**: technical debt items are divided into categories. For instance, severity levels: low, medium, and high risk;
- **Ordered List**: technical debt items are placed in a list and ordered by which payment should be made first. That is, the first items on the list should be the first ones to be paid off.

TD Item	Pay?	Category	Order
Item 42	Yes	High	1
Item 36	Yes	Medium	2
Item 8	Yes	Medium	3
Item 17	No	Low	4
Item 52	No	Low	5

Table 3.4: *Technical debt prioritization return example*

Table 3.4 shows an example of technical debt prioritization methods returns. The column "Pay?" indicates if a technical debt should (Yes) or not (No) be paid. The column Category indicates which payment category the item belongs to. The column Order indicates the payment order, i.e., Item 42 should be paid before Item 36, and so on.

Table 3.4 shows the result types for each selected paper. Note that for most methods, only one column will be filled. For example, Guo and Seaman (Yuepu Guo and Seaman, 2011) indicates if an item should be paid, but it does not indicate a category or an order. On the other hand, Schmid (Fontana *et al.*, 2015) uses Category and Ordered List, and Zazworka et al. (Zazworka, Seaman, and Shull, 2011) uses Boolean and Ordered List.

	Method Response			Method Characteristics			
Paper	Bool	Cat	List	CA	LI	Var	Tool
(Brauer *et al.*, 2017)		X	X	C	C	C	
(Schmid, 2013)	X				C		
(Fontana *et al.*, 2015)			X	C	C	p	
(Skourletopoulos, Chatzimisios, *et al.*, 2015)		X	X		p		
(Akbarinasaji, 2015)			X	C	C	C	
(Mohan *et al.*, 2016)		X	X	C	C	p	
(Codabux and Williams, 2016)	X			p	C		
(Zazworka, Seaman, and Shull, 2011)			X	C	C		
(Snipes *et al.*, 2012)	X		X	C	C		
(Falessi and Voegele, 2015)	X			C	p	p	p
(Chatzigeorgiou *et al.*, 2015)			X	C	p		
(Skourletopoulos, Mavromoustakis, *et al.*, 2016)			X		C		
(Yli-Huumo *et al.*, 2016)			X	C	C	p	
(Choudhary and P. Singh, 2016)	X		X	C	C	p	
(Charalampidou *et al.*, 2017)	X			p	C	p	
(L. F. Ribeiro *et al.*, 2017)			X	C	C	p	
(Haendler *et al.*, 2017)		X		p	C	p	
(Mensah *et al.*, 2018)		X		p	C		
(Plösch *et al.*, 2018)	X			C	C	p	
(Yuepu Guo and Seaman, 2011)	X			C	C	p	
(Fernández-Sánchez, Díaz, *et al.*, 2014)	X			C	C	p	
(Abad and Ruhe, 2015)	X			C	C	p	
(Aldaeej and Seaman, 2018)		X	X	C	C	p	
(Albarak and Bahsoon, 2018)			X	C			
(Tornhill, 2018)	X			p	p		p
(Alfayez and Boehm, 2019)			X	C	C	p	
(Kouros *et al.*, 2019)			X	C	C	p	

Table 3.5: *Technical debt prioritization methods response types and characteristics*

3.2.5 RQ3: What are the characteristics of the technical debt prioritization methods?

We extract the characteristics of the technical debt prioritization on real software projects by analyzing the papers:

- Context Adaptive (CA), that is, it could be used in different projects varying, for example, the domain, the methodology of development, the size, and organization of the development team and technologies;
- Language Independent (LI) to accept multiple programming languages;
- To cover a variety of technical debt types;
- Integrate to a tool that can be easily added into the software development workflow.

Table 3.5 shows each paper its categories. The capital 'C' indicates that it covers completely, while the letter 'p' indicates partial coverage.

Most methods are context-adaptive, i.e., they use variables in the prioritization methods to express the software context. For example, Fontana et al. (Fontana *et al.*, 2015) use code metrics to detect code smells.

Almost all papers present methods that could be applied to analyze independent programming languages. For example, Mohan et al. (Mohan *et al.*, 2016) present a reduction using search-based automated refactoring that could be applied to analyze technical debt in any programming language.

More than half of the papers cover a variety of technical debt types. However, only Akbarinasaji (Akbarinasaji, 2015), Bräuer, and Plösch (Brauer *et al.*, 2017) could cover many types. The first one uses a recommender system to prioritize the technical debt. The second one uses a two-dimensional portfolio matrix to analyze violations of rules. The other methods, such as (Yli-Huumo *et al.*, 2016; Charalampidou *et al.*, 2017; Falessi and Voegele, 2015) apply the method in one or a limited group of technical debt types.

Only two prioritization methods were integrated into a tool. However, both are partially integrated with a tool. For instance, Skourletopoulos et al. (Skourletopoulos, Chatzimisios, *et al.*, 2015) developed a plugin for SonarQube called Technical Debt Analyzer. The plugin collects data from several software releases using SonarQube, Issue Tracker, and version control systems. The data are statistically analyzed using R and WEKA, and with the results, the tool relates density quality violations with defect-prone.

3.3 Discussion of Findings

In this section, we present a discussion of the results. This systematic mapping review aimed to identify the methods for prioritizing technical debt. Therefore, through the mapping, it was possible to contribute to the discussion about how software development teams may prioritize their technical debt. Contributions from the Conceptual/Introductory category include practical considerations and challenges about how to handle technical debt, requirements for future technical debt tools, tools problems, metaphor dissemination and understanding, and possible results of prioritizing technical debt.

Besides discussing conceptual aspects involved in technical debt prioritization, some papers also try to define ways to standardize technical debt prioritization. Prioritization is done regardless of the method used, thus developing technical debt prioritization frameworks.

By analyzing the methods' approach and techniques presented in the papers, we built a two-level taxonomy to organize and relate the methods found in the literature. That taxonomy helps to find the methods that have already been explored in literature, and it also helps to find gaps in order to try applying new methods to prioritize technical debt.

We also found that the prioritization methods provide three responses: Boolean, to pay off the technical debt or not; Categories, which indicates how priority the payment is; Ordered List, sorted by prioritization order.

Finally, we analyzed practical characteristics in which prioritization methods should be applied to real software projects. We found out that any method is, at the same time: context-adaptive, i.e., it could be used in any software development context; language-independent, i.e., it could be applied to several programming languages; able to cover a variety of TD types; and to be fully integrated into a tool.

Despite all efforts to prioritize technical debt, there is no practical method to prioritize many technical debt types in different programming languages and contexts while integrating them into a tool that allows use in practice. In addition, most solutions at some point during prioritization need manual support to classify items and to define which should be paid for in the decision-making process.

3.4 Threats to Validity

The results of this systematic mapping study may be affected by bias in paper identification, study selection, data extraction, and data synthesis. In this section, we will discuss these biases.

Identification Bias Some factors can be difficult to ensure, as all relevant papers are in the search results. The choice of the publication venue may exclude some papers, so we considered the main digital libraries to minimize this threat. Besides that, the term “technical debt" in the search query may exclude relevant studies published before the term was widely used. The term may also exclude papers that only use "debt" or another similar term.

Selection Bias The definition of the inclusion and exclusion criterion can affect the selection of papers. To minimize this threat and remove personal bias, all authors revised each criterion.

Data Extraction Bias The keywording process may result in the inaccuracy of the extracted data keywords, which may affect the classification schema and data analysis. In order to reduce this bias, each author extracts a keyword set for each paper. Then, the keywords were compared and joined to form the final keyword set.

Data Synthesis Bias In some papers, all the required information was not directly available or sometimes needed to be clarified. Therefore, we needed to clarify some hidden information during the synthesis. Besides that, personal bias may affect the synthesis result. Each author performed a synthesis of the data to minimize this threat. Then the synthesis was compared, the common synthesis was joined, and the discrepant synthesis was evaluated to be included or excluded.

3.5 Mapping Conclusion

This study performed a systematic literature mapping following Petersen et al. (Petersen *et al.*, 2008). We searched for relevant technical debt prioritization studies in six main computing databases: ACM Digital Library, IEEE Xplore, Science Direct, Scopus, Springer Link, and Web of Science. Finally, we selected 51 unique papers.

As a result, we built a technical debt prioritization taxonomy with two levels and ten categories. In the first level, categories are directly related to the approaches of the papers. In the second level, categories are derived from the first level and aim for specific aspects addressed in the papers.

Most of the methods can adapt to the contexts, that is, to adapt to different development methodologies, different numbers of team members, and technologies. In addition, most of the methods do not need a specific programming language to work or could be applied to several programming languages. On the other hand, half of the methods are not able to deal with several technical debt types, and only one method is integrated with the development tools.

Therefore, as discussed in Section 3.3, the methods found in the literature do not cover all practical aspects for technical debt prioritization to be used in real projects. For example, most of the methods are not implemented in a tool or cover only one programming language.

The main contributions of the mapping literature review are:

- Two-level taxonomy;
- A collection of technical debt prioritization methods;
- Technical debt prioritization result types: boolean, category and ordered list;
- Practical methods characteristics: context-adaptive, language-independency, several technical debt types, and tool integration.

We also identified some research gaps: methods should consider the development context methods that work for several programming languages, cover the prioritization of many technical debt types, and be integrated into a tool to evaluate them in practice.

Future work should focus on developing a method that could be used in practice. From this taxonomy and the mapping of the characteristics used in each prioritization method, it is possible to combine different methods to cover a more significant number of technical debt types and automate their prioritization, such as with artificial intelligence. This will facilitate and streamline the method application in software projects as an aid tool in the prioritization process and consequent technical debt management.

3.6 Related Work for Technical Debt Prioritization Criteria Study

Technical debt prioritization study 5 is a qualitative Grounded Theory study to understand how developers prioritize technical debt items in real software projects.

We collected developers' opinions about whether and when to pay a technical debt item and why they made that decision. We used InteraSurveyTD 4.2 to show only technical debt items from software projects they have contributed to.

Our qualitative study extends Becker's (Becker *et al.*, 2018) theoretical process by studying how prioritization decisions are made in practice. By identifying decision criteria, our study also builds on Leppanen et al.'s framework (Leppanen *et al.*, 2015).

3.7 Related Work for Technical Debt Prioritization using Machine Learning

Most of the papers presented in this section are related to our quantitative study that aims to develop machine learning methods to prioritize technical debt items payment. In particular, studies in the Mathematical/Statistical, Code Metrics, and Financial categories.

Tsoukalas et al. (Tsoukalas, Mittas, *et al.*, 2021) used well-known machine learning methods to classify software modules in high-TD or not. Our work uses almost the same machine learning methods, but we applied them to classify whether and when a technical debt item should be paid off.

Chapter 4

Sonarlizer Xplorer and InteraSurveyTD

We have developed two tools to support our upcoming studies: Sonarlizer Xplorer and InteraSurveyTD. Sonarlizer Xplorer is a tool to mine public GitHub repositories and to identify a large number of technical debt items and code metrics. InteraSurveyTD is a tool to show developers technical debt items from projects they have contributed to and ask them whether/when the item should be paid off and optionally why they made the decision.

4.1 Sonarlizer Xplorer

Sonarlizer Xplorer is a tool to mine and analyzes public GitHub projects resulting in a dataset with many technical debt items and code metrics. It also includes a list of GitHub repositories, users, and organizations.

The mining process starts with a repository. The tool finds users and organizations through the GitHub API. It then finds new repositories using the repositories already found. Then the tool analyzes each project using SonarQube to identify technical debt items and extract code metrics.

The primary users of Sonarlizer Xplorer tools are researchers who need large amounts of data from software projects to use in their investigations, particularly technical debt items and code metrics. Such a need for large data sets is often associated with using machine learning as part of the research design.

4.1.1 Use Cases

Sonarlizer Xplorer can collect data and analyze public GitHub projects, that is, to calculate statistics for technical debt items and code metrics. For example, we can use it to calculate the mean, median, and standard deviation of the project's lines of code. Alternatively, it can also compute the average of a specific technical debt type on projects with certain characteristics, such as size (e.g. "on average, how many high-complexity classes are found in files with less than 1,000 LOC?").

Our tool can also be handy when finding public software projects hosted on GitHub and their respective developers. After analyzing the SonarQube project, we sent an interactive survey to the developers, asking when a technical debt item should be paid off. We used the code metrics collected by the tool and the survey responses to train and evaluate machine learning methods to try to prioritize the payment of technical debt items.

4.1.2 Architecture, Technologies, and Implementation

Sonar Xplorer consists of two interconnected sub-tools. First, it calls GitHub Xplorer to walk through the GitHub API mining public repositories, users, and organizations to store them in a MongoDB database. Then, the tool calls Sonarlizer to analyze each repository using SonarQube to identify technical debt items and extract code metrics.

Sonarlizer Xplorer is developed in Node.JS and uses MongoDB database to handle large volumes of non-relational data. In addition, all SonarQube data, including technical debt items and code metrics, are stored in PostgreSQL. We chose these technologies to allow distributed computing, large amounts of data, and a simple tool installation. Figure 4.1 shows the tool architecture, where the flow starts on mining GitHub and finishes with technical debt items and code metrics stored in a PostgreSQL database.

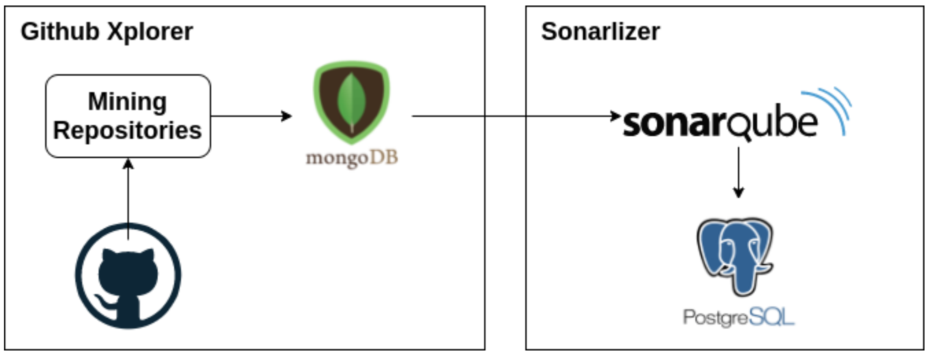


Figure 4.1: *Sonarlizer Xplorer architecture.*

4.1.3 GitHub Xplorer

GitHub Xplorer is a tool for mining public GitHub repositories to find real software projects and their related developers and organizations. A repository is a virtual place to store software project source code. No public dataset is available containing GitHub repositories, organizations, and developers' information. For this reason, we used GitHub Xplorer to collect the data to build this dataset.

Finding Repositories, Developers, and Organizations

A repository is an entity that represents a project codebase, and it is related to many users (developers) and at most one organization (a group of developers). Examples of repositories include Apache Maven [1], Google Kubernetes [2], Microsoft Visual Studio Code [3], and TensorFlow [4]. Users can contribute to many repositories. They can also participate in many organizations. Examples of organizations include Apache Foundation [5], Microsoft [6], and Kubernetes [7]. Users and organizations can own repositories, but a repository has only one owner (user or organization). Figure 4.2 shows GitHub data entities and the relationships among them.

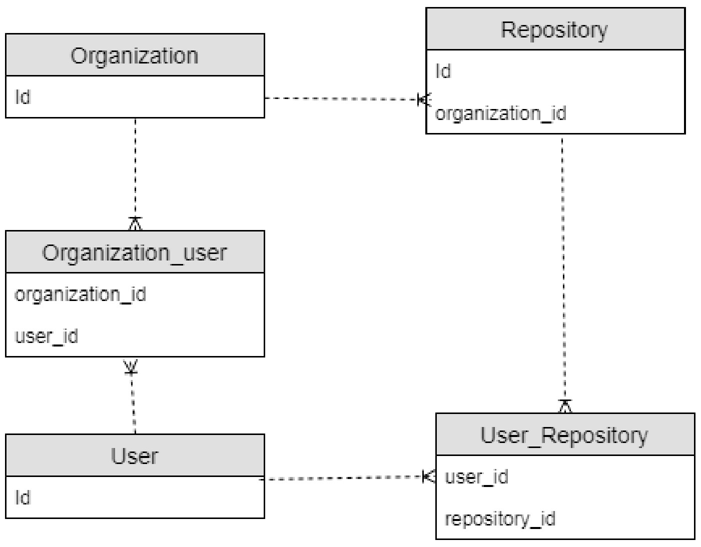


Figure 4.2: *GitHub data entities and relationships.*

We used the GitHub REST API [8] to access public GitHub data. We implemented a feature that uses a list of access tokens to parallelize the data requests and speed up the mining process.

Each GitHub API call returns only one repository, organization, or user (developer). For this reason, we had to use entity relationships to find all the data. For example, when we request a repository, we can access the list of developers who contribute to it and the owner organization if it exists. When we request an user, we can access the list of repositories they contribute to and the list of organizations to which they belong.

GitHub Xplorer uses three sets (lists) to store entities to be processed: repositories, organizations, and users. For each repository, the tool gets the entity data, list of users, and organizations (if an organization is the repository owner); thus, each repository request can find new users and one organization. For each user, the tool collects user data, its repositories, and organizations; thus, each user request can find new repositories and

[1] https://github.com/apache/maven

[2] https://github.com/kubernetes/kubernetes

[3] https://github.com/microsoft/vscode

[4] https://github.com/tensorflow/tensorflow

[5] https://github.com/apache

[6] https://github.com/microsoft

[7] https://github.com/kubernetes

[8] https://docs.github.com/pt/rest

organizations. For each organization, the tool gets organization data, its repositories, and users; thus, each organization request can find new users and repositories.

The following snippet shows the pseudo-code for processing the repository set. The Git Xplorer similarly treats user and organization sets.

```
SET repositories_to_process, users_to_process, organizations_to_process;
SET repositories_set, users_set, organizations_set;

begin parallel:
while repositories_to_process is not empty
   repository = pop repositories_to_process
   getGitHubInfo(repository)
   addUpdate(repository, repositories_set)

   users = getGitHubUsers(repository)
   push(users, users_to_process)

   organization = getGitHubOrganizations(repository)
   push(organizations, organizations_to_process)
end parallel
```

Note that a user or organization is stored in the set only if they we not on the processed or to-be-processed list. This process ensures that an entity is not processed more than once, avoiding repetitions and wasted time and processing.

The data mining starts by adding a GitHub repository to the repository set. The tool consumes the user and organization lists iteratively to get the complete entity information and related entities that have not yet been processed to feed the queues.

On Sonarlizer Xplorer, each repository is stored in a document in a MongoDB collection. The *status* property identifies when the project was (status = 1) or not (status = 0) analyzed. Figure 4.3 shows an example of a repository document in MongoDB.

4.1.4 Sonarlizer

Sonarlizer automatically performs a SonarQube analysis to identify technical debt items and code metrics in the public software projects hosted on GitHub and discovered by GitHub Xplorer. For Java projects, Sonarlizer needs to compile the source code before analyzing it. For that, the tool identifies the builder among the main ones for Java: Apache Maven[9], Apache Ant[10], or Gradle[11]. Then, it uses the build-specific commands for these tools to compile and send the compiled files to SonarQube to analyze. If no builder is identified, the standard SonarQube command is performed through the SonarScanner[12] tool.

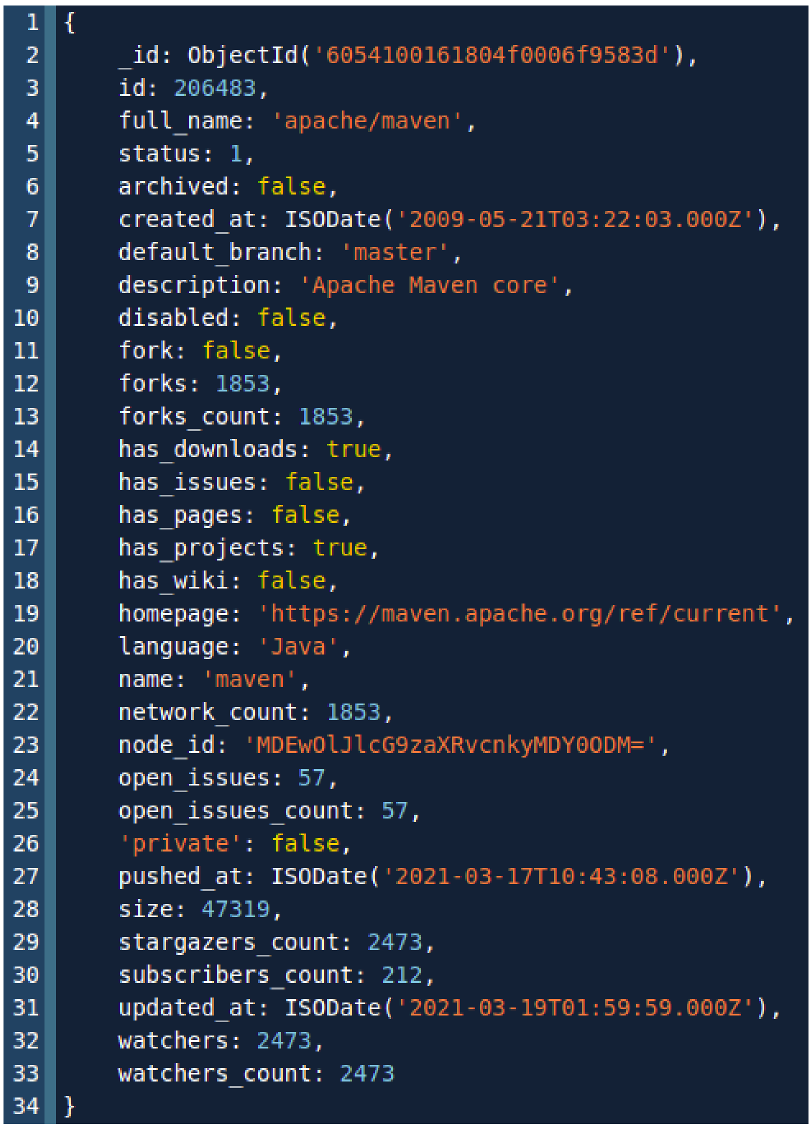
As input, Sonarlizer queries the MongoDB repository collection to retrieve the not analyzed GitHub repository data. Sonarlizer clones the repository from GitHub to a local

[9] https://maven.apache.org

[10] https://ant.apache.org

[11] https://gradle.org

[12] https://docs.sonarqube.org/latest/analyzing-source-code/scanners/sonarscanner



```
{
    _id: ObjectId('6054100161804f0006f9583d'),
    id: 206483,
    full_name: 'apache/maven',
    status: 1,
    archived: false,
    created_at: ISODate('2009-05-21T03:22:03.000Z'),
    default_branch: 'master',
    description: 'Apache Maven core',
    disabled: false,
    fork: false,
    forks: 1853,
    forks_count: 1853,
    has_downloads: true,
    has_issues: false,
    has_pages: false,
    has_projects: true,
    has_wiki: false,
    homepage: 'https://maven.apache.org/ref/current',
    language: 'Java',
    name: 'maven',
    network_count: 1853,
    node_id: 'MDEwOlJlcG9zaXRvcnkyMDY0ODM=',
    open_issues: 57,
    open_issues_count: 57,
    'private': false,
    pushed_at: ISODate('2021-03-17T10:43:08.000Z'),
    size: 47319,
    stargazers_count: 2473,
    subscribers_count: 212,
    updated_at: ISODate('2021-03-19T01:59:59.000Z'),
    watchers: 2473,
    watchers_count: 2473
}
```

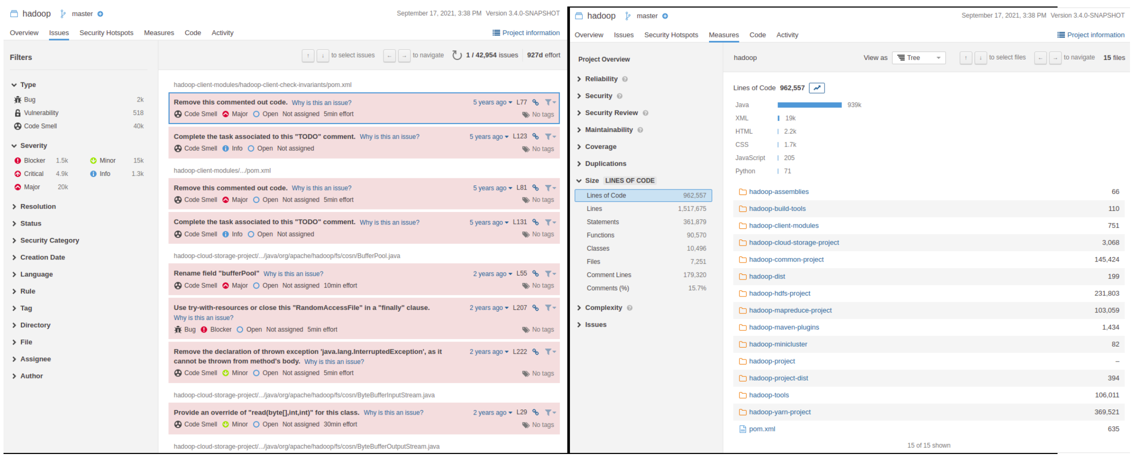
Figure 4.3: *Repository document example.*

machine. Then it performs a SonarQube analysis to identify technical debt items such as naming convention violations, high code complexity, and large code files; and code metrics, such as number of lines, number of files, complexity average by file, and cognitive complexity. SonarQube has a list of rules that identify technical debt items, and when one of these rules is broken, an issue is created. Some examples of rules are *Member Name, Unused Imports, Nested If Depth, and Method Length.*

Finally, Sonarlizer goes through the commit list to relate each source file to the users who have contributed to it.

4.1.5 Results

As a result, Sonarlizer Xplorer provides a list of technical debt items and code metrics for many public GitHub repositories. Figure 4.4 shows an issue list example on SonarQube and a code metrics list example on SonarQube.



September 17, 2021, 3:38 PM Version 3.4.0-SNAPSHOT

1 / 42,954 issues 927d effort

Type	
Bug	2k
Vulnerability	518
Code Smell	40k

Severity			
Blocker	1.5k	Minor	15k
Critical	4.9k	Info	1.3k
Major	20k		

Resolution
Status
Security Category
Creation Date
Language
Rule
Tag
Directory
File
Assignee
Author

hadoop-client-modules/hadoop-client-check-invariants/pom.xml

Remove this commented out code. Why is this an issue? 5 years ago L77
Code Smell Major Open Not assigned 5min effort No tags

Complete the task associated to this "TODO" comment. Why is this an issue? 5 years ago L123
Code Smell Info Open Not assigned No tags

hadoop-client-modules/.../pom.xml

Remove this commented out code. Why is this an issue? 5 years ago L81
Code Smell Major Open Not assigned 5min effort No tags

Complete the task associated to this "TODO" comment. Why is this an issue? 5 years ago L131
Code Smell Info Open Not assigned No tags

hadoop-cloud-storage-project/.../java/org/apache/hadoop/fs/cosn/BufferPool.java

Rename field "bufferPool" Why is this an issue? 2 years ago L55
Code Smell Major Open Not assigned 10min effort No tags

Use try-with-resources or close this "RandomAccessFile" in a "finally" clause. Why is this an issue? 2 years ago L207
Bug Blocker Open Not assigned 5min effort No tags

Remove the declaration of thrown exception 'java.lang.InterruptedException', as it cannot be thrown from method's body. Why is this an issue? 2 years ago L222
Code Smell Minor Open Not assigned 5min effort No tags

hadoop-cloud-storage-project/.../java/org/apache/hadoop/fs/cosn/ByteBufferInputStream.java

Provide an override of "read(byte[],int,int)" for this class. Why is this an issue? 2 years ago L29
Code Smell Minor Open Not assigned 30min effort No tags

hadoop-cloud-storage-project/.../java/org/apache/hadoop/fs/cosn/ByteBufferOutputStream.java

Project Overview
Reliability
Security
Security Review
Maintainability
Coverage
Duplications
Size LINES OF CODE

Lines of Code	962,557
Lines	1,517,675
Statements	361,879
Functions	90,570
Classes	10,496
Files	7,251
Comment Lines	179,320
Comments (%)	15.7%

Complexity
Issues

hadoop 15 files

Lines of Code 962,557

Java	939k
XML	19k
HTML	2.2k
CSS	1.7k
JavaScript	205
Python	71

hadoop-assemblies	66
hadoop-build-tools	110
hadoop-client-modules	751
hadoop-cloud-storage-project	3,068
hadoop-common-project	145,424
hadoop-dist	199
hadoop-hdfs-project	231,803
hadoop-mapreduce-project	103,059
hadoop-maven-plugins	1,434
hadoop-minicluster	82
hadoop-project	–
hadoop-project-dist	394
hadoop-tools	106,011
hadoop-yarn-project	369,521
pom.xml	635

15 of 15 shown

Figure 4.4: *Left: Example of technical debt items on SonarQube. Right: Example of code metrics on SonarQube.*

We used the tool from July 2021 to October 2021 to mine GitHub. We found 57,382,956 repositories and 4,430,010 users. Table 4.1 shows the number of projects found by programming language.

Language	# of projects
JavaScript	9,867,584
Python	5,372,460
Java	4,777,964
Ruby	2,031,174
C++	1,993,982
C	1,468,370
C#	1,466,983

Table 4.1: *Example of project quantities mined by programming language.*

In parallel with mining, we also extracted 609,884 Java projects where SonarQube successfully analyzed 45,994 (about 7.5%). The low rate is because to analyze Java code on SonarQube, it is first necessary to compile projects. Even using a builder, most projects require specific Java versions, configuration files, or additional parameters to compile successfully. Table 4.2 shows examples of the number of technical debt items by severity and projects by *ncloc*.

Severity	# TD items		ncloc	# Projects
Blocker	467k		<1k	27k
Critical	2.4M		1k - 10k	14k
Major	5.2M		10k - 100k	4.6k
Minor	6.6M		100k - 500k	390
Info	400k		>500k	17

Table 4.2: *Number of technical debt items by severity and projects divided by a non-comment line of code (ncloc).*

4.1.6 Related Tools

Most GitHub mining tools can mine data from a given repository, such as RepoDriller (Aniche, 2012), which extracts commits, developers, modifications, diffs, and source code. GH Torrent (Gousios, 2013) allows querying of GitHub events from public repositories. GH Crawler (*Microsoft GHCrawler* n.d.) is a Microsoft project that finds new repositories, although it was created to retrieve all GitHub entities related to an organization, repository, user, and team.

All these tools analyze just one project to identify technical debt items and code metrics. SonarQube (Campbell and Papapetrou, 2013) is one of the tools that can do that. Findbugs (Ayewah *et al.*, 2008) finds bugs and technical debt items and classifies them according to their severity. PMD (Araujo *et al.*, 2011) runs a cross-language static code analysis to find technical debt items.

Although there are many tools to mine a given GitHub repository, and some can be used to identify technical debt items, they need to find the repositories and analyze them to identify technical debt items and extract code metrics.

4.1.7 Future Enhancements

The tool can be customized to get more data provided by the GitHub API, such as commit files, modifications, and comments; GitHub issues; pull requests; events; and topics. Adding more SonarQube extensions to analyze more aspects such as test coverage, security vulnerability, code metrics, and more technical debt types is also possible.

A first improvement could be adding new repository integration to GitHub Xplorer, for example, to allow for mining public projects on Gitlab[13], BitBucket[14] and Source-Forge[15].

On the other hand, to increase the data types provided by the tool, the GitHub Analyzer tool could collect other metrics and data related to the hosted project, such as commits and releases. Besides that, Sonarlizer could use other tools to analyze the code to provide more code metrics and find new technical debt items, such as FindBugs[16] and PMD[17].

Sonarlizer can also identify technical debt items using other tools such as Cast [18] and Squore [19].

[13] https://www.gitlab.com

[14] https://www.atlassian.com/software/bitbucket

[15] https://sourceforge.net

[16] http://findbugs.sourceforge.net

[17] https://pmd.github.io

[18] https://www.castsoftware.com

[19] https://www.vector.com/int/en/products/products-a-z/software/squore

4.1.8 License

The Sonarlizer Xplorer is licensed under the MIT License, a short and simple permissive free software license to any person obtaining a copy of this software and associated documentation files[20]. More details can be found in the license file[21].

4.2 InteraSurveyTD

We developed the customer survey tool InteraSurveyTD to show developers technical debt items from projects they have contributed and ask them whether and when the item should be paid off, and then explain the decision. We sent invitation emails [22] to the developers with a project brief and a link to the questionnaire with an identifier to load the technical debt items specific to that developer. Once the developer(respondent) follows the link, they are shown a set of instructions for completing the survey and informed consent for participation [23]. Before starting the survey, they had to confirm that they read and agreed with the consent terms and that to be at least 18 years old. Then, the InteraSurveyTD tool randomly picked a technical debt item up from all the items of the respondent's project.

To preserve anonymity, not all the items shown to a respondent are from files they have contributed to. InteraSurveyTD randomly chooses an item specific to the developer respondent only 70% of the time. This avoids a particular file with just one contributor, so the developer's responses could be identified. Also, we do not store information about any relationship between the answer and the participant. All selected projects have three or more participants, which allows for anonymity and prevents tracking of participant answers.

Figure 5.1 shows a survey screen presenting a technical debt item and the questions that capture the developer's view of when the technical debt item must be paid off.

Question 1 has six possible answers on a descending scale according to how urgent the item is (these explanations are provided in the instructions to respondents):

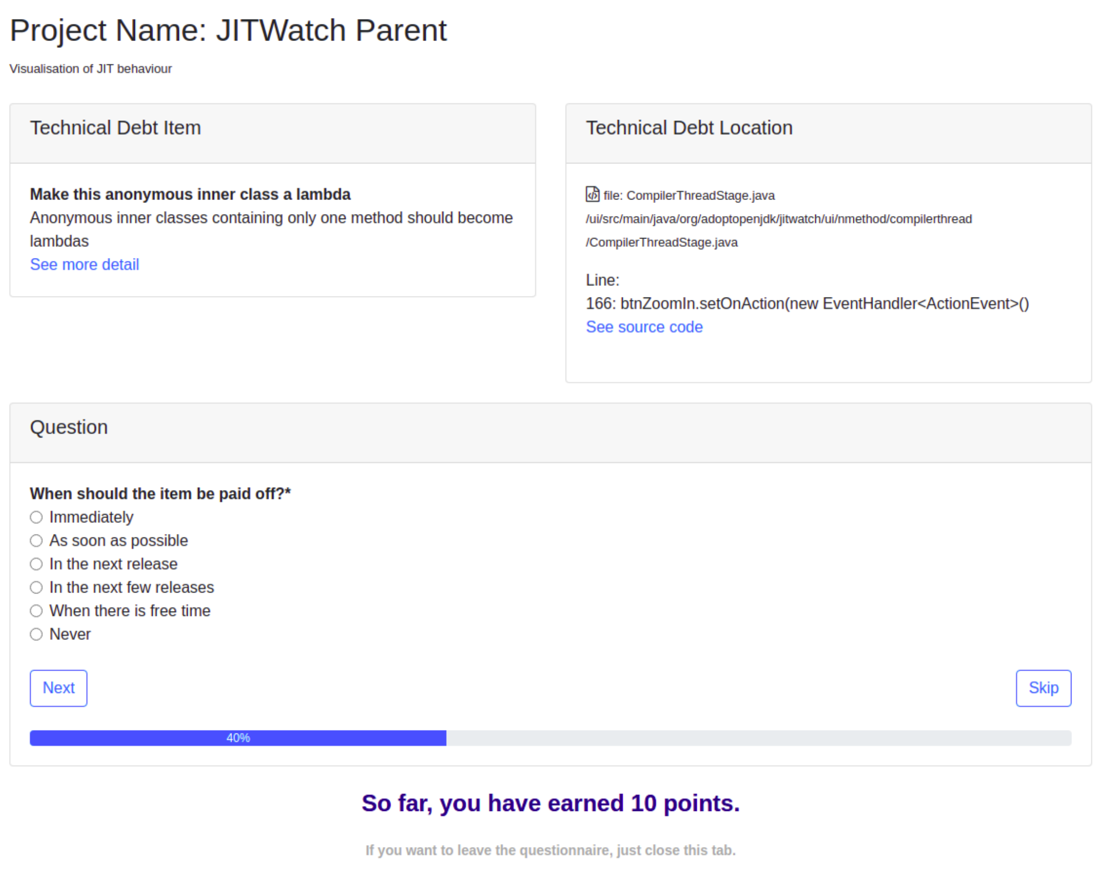
- **Immediately**: pay the item off before developing anything else;
- **As soon as possible**: pay item off in the current release;
- **In the next release**: plan item payment for next release;
- **In the next few releases**: it doesn't postpone payment indefinitely, but it doesn't have to happen in the next release;
- **When there is free time**: no planning is required, but eventually the item should be paid;

[20] https://github.com/git/git-scm.com/blob/main/MIT-LICENSE.txt

[21] https://github.com/diogojpina/sonarlizer-xplorer/blob/master/LICENSE

[22] https://zenodo.org/record/6384731/files/email-template-anonymized.pdf?download=1

[23] https://zenodo.org/record/6384731/files/research-web-consent-anonymized.pdf?download=1


Project Name: JITWatch Parent

Visualisation of JIT behaviour

Technical Debt Item

Make this anonymous inner class a lambda
Anonymous inner classes containing only one method should become lambdas
See more detail

Technical Debt Location

file: CompilerThreadStage.java
/ui/src/main/java/org/adoptopenjdk/jitwatch/ui/nmethod/compilerthread/CompilerThreadStage.java

Line:
166: btnZoomIn.setOnAction(new EventHandler<ActionEvent>()
See source code

Question

When should the item be paid off?*
- Immediately
- As soon as possible
- In the next release
- In the next few releases
- When there is free time
- Never

Next Skip

40%

So far, you have earned 10 points.

If you want to leave the questionnaire, just close this tab.

Figure 4.5: *Questionnaire question example.*

- **Never**: the item is not important for the project or it is not in fact a technical debt item, or for some other reason, should not be paid.

The answer to question 2 is an open text field where the respondents can explain their answer to question 1. After storing the answers to both questions, InteraSurveyTD repeats the process with another TD item.

Respondents can leave the survey at any time, and come back using the email invitation link. To motivate respondents, there was also a gamification scheme whereby developers could earn points by answering more questions and by referring other developers to the study. Points earned allowed respondents to receive prizes at the end of the study.

4.3 Conclusion

Sonarlizer Xplorer is a tool to mine public GitHub repositories and analyze them using SonarQube to identify technical debt items and code metrics. We describe how the tool explores GitHub through its API to find a list of public repositories related to an initial repository. Then it runs a SonarQube analysis to extract a list of technical debt items and a list of code metrics.

We also show possible applications of the tool in research using conventional statistics and more complex methods such as machine learning to analyze and predict patterns and behaviors for technical debt prioritization.

The tool is in its first version. Thus, several features can be implemented, such as mining other code repository host platforms, applying other code analysis tools, and collecting metrics from code repositories.

Sonarlizer Xplorer is a tool that researchers can use to increase the number of technical debt items for analysis, in order to derive more robust conclusions and find new approaches and behaviors that cannot be found by analyzing a few projects.

InteraSurveyTD is an interactive survey tool to show technical debt items to which the developer contributed and collect two answers: when the item should be paid off and why.

Chapter 5

Technical Debt Prioritization Criteria

In this chapter, we aim to understand which criteria software developers use in practice to prioritize code technical debt items in real software projects. This fills a gap in the literature regarding studies of technical debt prioritization criteria in real software projects.

We used a survey to collect information from developers about open-source projects hosted on GitHub. The survey questions were based on code technical debt items on projects to which the respondent had contributed. After showing each item, the survey asks the respondent to indicate how soon the item should be paid off and why.

We analyzed the answers using Straussian Grounded Theory (Straussian GT) techniques, namely open coding, axial coding, and selective coding, to identify the criteria developers used to prioritize code technical debt. We grouped the criteria into 15 categories and 2 super-categories related to paying off the technical debt item: CODE_IMPROVEMENT and COST_BENEFIT; and 3 super-categories related to not paying off the item PROJECT_SPECIFIC_DECISION, PROBLEM_WITH_RULE, and UNUSED_CODE.

We found that when developers chose to pay off a code debt item, they decided to do so soon. When they chose not to pay it was a project-specific decision. Also, when developers used the same criterion, the payment priority chosen was in the same neighborhood. Finally, we noted that each project needs a specific set of criteria to prioritize technical debt.

This work addresses the following research questions:

- RQ1. How do developers prioritize technical debt?
    - RQ1.1. How do developers decide whether a code technical debt item should or should not be paid off?
    - RQ1.2 How do developers decide when a code technical debt item should be paid off?

5.1 Research Method

In this study, our primary goal is to understand which criteria software developers use to decide whether and when a code technical debt item should be paid off in real software projects. For that purpose, we decided to use a survey to reach many projects and multiple types of code debt rather than applying interviews that would bring more details but for a smaller number of projects.

We found public Java repositories from Github and analyzed them with SonarQube. Then we sent a survey to participants asking two questions for each technical debt item presented. The first is a multiple-choice question on when the item payment should be made. The second is an open-text question to explain the reason for the priority choice in the first question. These two questions provided data to understand how developers decide whether and when a code debt should or not be paid off. Then we analyzed the data using Straussian Grounded Theory.

5.1.1 Data Collection

We used Sonarlizer Xplorer 4.1 tool to scrap public GitHub software code repositories to find a large number of Java open-source projects and their developers. We also used the tool to analyze the projects with SonarQube, resulting in code technical debt items related to each project. An example of such items is *"Remove this "close" call; closing the resource is handled automatically by the try-with-resources from the HttpProxy repository"*. We proceeded to the next step from the identified technical debt items, asking developers to evaluate those items via our survey.

We used the InteraSurveyTD 4.2 tool to manage and apply the survey. It shows respondents only those technical debt items related to the projects they have worked on. The tool shows a technical debt item and related information, such as a file, line location, and description. Then the respondent is asked, "When should the item be paid off?" (multiple-choice) and "Why?" (open-text field).

The data extraction started by adding the Apache Maven repository to the repository queue. The tool consumes the developer, organization, and repository queues iteratively to get the complete entity information and their related entities that have not yet been processed to feed the queues.

Survey

We used the InteraSurveyTD 4.2 tool to show developers technical debt items from projects they have contributed to and ask them whether and when the item should be paid off.

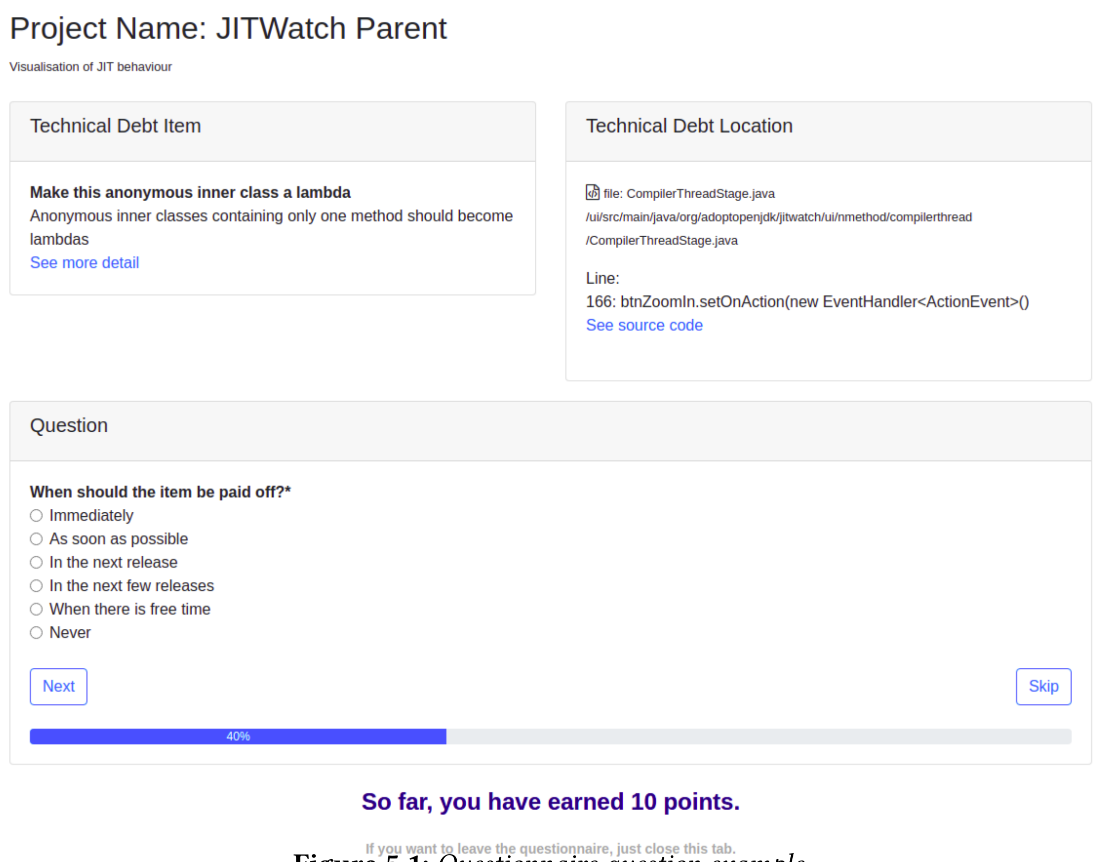
We sent invitation emails [1] to the developers with a project brief and a link to the survey with an identifier to load the technical debt items specific to that developer. Once the developer (respondent) follows the link, they are shown a set of instructions for completing

[1] https://zenodo.org/record/6384731/files/email-template-anonymized.pdf?download=1

the survey and informed consent for participation [2]. Before starting the survey, they had to confirm, read and agree with the consent terms and be at least 18 years old. Then, the InteraSurveyTD tool randomly chooses a technical debt item from all the items of the respondent's project.

As detailed in Section 4.2, not all the items shown to a respondent are from files they have contributed to, preserving anonymity. Avoiding a particular file has just one contributor, so the developer's responses could be identified. Also, we do not store information about any relationship between the answer and the participant. All selected projects have three or more participants, which allows for anonymity and prevents tracking of participant answers.

Figure 5.1 shows a survey screen presenting a technical debt item and the questions, which capture the developer's view of when the technical debt item needs to be paid off.



Project Name: JITWatch Parent

Visualisation of JIT behaviour

Technical Debt Item

Make this anonymous inner class a lambda
Anonymous inner classes containing only one method should become lambdas
See more detail

Technical Debt Location

file: CompilerThreadStage.java
/ui/src/main/java/org/adoptopenjdk/jitwatch/ui/nmethod/compilerthread/CompilerThreadStage.java

Line:
166: btnZoomIn.setOnAction(new EventHandler<ActionEvent>()
See source code

Question

When should the item be paid off?*
- ○ Immediately
- ○ As soon as possible
- ○ In the next release
- ○ In the next few releases
- ○ When there is free time
- ○ Never

Next | Skip

40%

So far, you have earned 10 points.

If you want to leave the questionnaire, just close this tab.

Figure 5.1: *Questionnaire question example.*

Question 1 has six possible answers on a descending scale according to how urgent the item is (these explanations are provided in the instructions to respondents):

- **Immediately**: pay the item off before developing anything else;
- **As soon as possible**: pay item off in the current release;
- **In the next release**: plan item payment for next release;

[2] https://zenodo.org/record/6384731/files/research-web-consent-anonymized.pdf?download=1

- **In the next few releases**: it doesn't postpone payment indefinitely, but it doesn't have to happen in the next release;
- **When there is free time**: no planning is required, but eventually the item should be paid;
- **Never**: the item is not important for the project or it is not in fact a technical debt item, or for some other reason, should not be paid.

The answer to question 2 is an open-text field where the respondents can explain why they decided on that priority. After storing the answers to both questions, InteraSurveyTD repeats the process with another TD item.

Respondents can leave the survey anytime, and return using the email invitation link. There was also a gamification scheme whereby developers could earn points by answering more questions and by referring other developers to the study. Points earned allowed respondents to receive prizes at the end of the study.

Pilot Study

We conducted a pilot study to evaluate and improve the data collection flow. The pilot study included 15 students who developed open-source software in Java in an Extreme Programming course from the University of São Paulo. We sent the invites through email and Facebook Messenger with the link to the survey.

After one week, we sent an evaluation survey to understand how easy it was to respond to the original survey, if the website and invite email provided clear and complete information about the research and technical debt, and concerning the gamification. For each of these areas, we asked for improvement suggestions. Five students answered the evaluation survey.

We used the collected suggestions to improve parts of the text, add more information, improve the interface, and change the survey flow - for example, by adding a button to skip a question.

5.1.2 Data Analysis

We applied Straussian GT Strauss and Corbin, 1994; Anselm Strauss and Juliet Corbin, 1998 to analyze the survey data qualitatively. We decided on Straussian GT because we defined the research questions upfront and derived from the literature Stol *et al.*, 2016, and they are broad and open-ended.

We started data analysis as soon as we had data available. We applied open, axial, and selective coding. Every time we added a new code, we would write a memo describing it. Otherwise, we tried to improve the existing one. We applied these techniques iteratively for each new response. We stopped collecting data when we identified that the answers did not produce new codes.

From the beginning of the data collection, we constantly compared data, memos, codes, subcategories, categories, and super-categories to ensure that the data were correctly interpreted and in the categories that best fit them. The time between sending the first invitation by e-mail and closing the survey was around six weeks.

We applied open coding to the survey responses. This process involves segmenting the answers into excerpts with a singular meaning and expressing that meaning with a code. For each answer to question 1, we applied a unique code based on the multiple-choice option. For example, for the answer "As soon as possible" we used the code ASAP. For answers to question 2, we applied between one and three codes for each answer. Like in "This code is correct. Lint rules cannot be applied blindly.", where we applied two codes: RULE_SHOULD_NOT_BE_APPLIED and LINT_RULES. That was an iterative process where we added, changed, and removed codes with each answer analyzed.

We reassembled the open codes in new ways to form categories during axial coding. Our goal was to create a higher abstraction level. Thus, we grouped codes to form subcategories, and in turn, we organized them into categories. In addition, we also tried to find relationships between the categories to form super-categories. This process was highly iterative, with codes and categories forming and re-forming as more data was incorporated into the evolving understanding. We wrote a memo to explain each category and provide examples of the answers that motivated its creation.

We applied selective coding to refine and integrate categories, revealing the main categories and indicating the developers' criteria to prioritize code debt and their relationships. We identified the technical debt prioritization criteria used by the developers from the categories and their relationships.

5.2 Results

In this section, we describe our results and findings. Although applying the Gaussian GT techniques has been made iteratively, and all the steps were taken simultaneously, in order to improve reading, we divided them into open, axial, and selective coding.

We sent invite emails to 2,471 developers distributed in 855 projects. From the total, 1,869 developers opened the invitation email, and 341 accessed the survey; however, only 39 developers from 21 projects [3] answered it.

The developers chose 11 times to pay off the technical debt item immediately, 30 times as soon as possible, 7 in the next release, and 66 never; thus, 42% chose to pay off the technical debt, and 58% chose not to pay it off.

5.2.1 Open Coding

We organized the collected data in a spreadsheet [4]. Each row contained one participant's answer related to one technical debt item. Besides the developers' answers, each row has columns to describe the technical debt item based on SonarQube data. All open codes were created in vivo, meaning they were derived directly from the collected data.

[3] https://zenodo.org/record/6384731/files/projects.csv?download=1

[4] Our data is available at https://zenodo.org/record/6384731/files/answers.csv?download=1

5.2.2 Axial Coding

We organized and assembled the open codes to form two levels of categories: the first level contains categories that group open codes, and the second level contains super-categories that group the first-level categories. For each open code, we tried to add it to an existing category that encompassed its meaning. We created a new category when it was not possible. We performed this step iteratively, constantly revisiting and evaluating the group of categories. We performed the same iterative process to group the categories into super-categories. Below we list and briefly discuss the categories and super-categories.

Super-Category PROJECT_SPECIFIC_DECISION includes the following categories:

DESIGN_DECISION: More experienced developers often use design patterns to create the software architecture and write the code. However, some of these design patterns break quality rules that generally apply to only a code snippet and not the architecture of the software as a whole. Therefore, it makes sense to break some quality rules to keep a design standard for some software architectures. Thus it is easier to read, maintain, scale, and improve the performance of the software.

When developers needed to choose between paying a technical debt item or not touching the design, they always chose to preserve the design and *Never* pay off the debt. One developer refers to another Lint rule: “This code is correct. Lint rules cannot be applied blindly”, and for another item, he referred to Javadocs to explain his decision: “This file is correct per the Javadoc documentation”.

MEANINGFUL_NAMES: Some developers prefer to use their naming conventions for software projects or modules. For example, Java convention defines variable names using the following regular expression: ’ˆ[a-z][a-zA-Z0-9]*$’, that is, the first character must be a lowercase letter followed by zero or more alphanumeric characters. Using other conventions triggers a TD item.

One developer explained that the names came from reflection: “I suspect this code has to interface with generated code that gets those names via reflection.” Another developer preferred using their naming convention: “I prefer the way I name variables/parameters to the Java convention”. Another one uses specific names for that project: “The name is specific to the project”. Another developer explained: “There are reasons for the names. I likely will never fix them.”

KEEP_READABILITY: Sometimes giving up standards to make the code more readable and easier to understand is the best choice. Some problem solutions are complex, and trying to reduce or fit them into code patterns can make the solution hard to read and understand, so it is best to leave the pattern aside so that the code is easier to maintain.

Two developers chose *Never* to pay off the technical debt items to keep the code easy to read: “GitException is used to carry failure information and declaring it makes it clear that it can be thrown.”, and “No. Code is more readable the way it is.”.

BREAK_SOMETHING_ELSE: Sometimes paying off a technical debt item could break something else; that is, changing a snippet could break functionality or compatibility.

For this reason, sometimes paying off a technical debt item is not worth it because its principal could be very high, costing several days of development.

One respondent chose to pay off an item *As soon as possible* because changing the code can break the existing code: “This TODO should be treated with more care, because it can break existing code.” For two technical debt items, another developer chose *Never* to pay off the technical debt items because: “Removing this code would break functionality and compatibility”. And for the other: “Deprecation in Jenkins does not mean "remove the code". If we remove the code, it will break compatibility. One of the compelling values of Jenkins is that it retains compatibility so that plugins compiled many years ago continue to operate with current releases.”

Super-Category PROBLEM_WITH_RULE includes the following categories:

RULE_SHOULD_NOT_BE_APPLIED: Some projects use their standards, so not all quality rules should apply.

Every time a respondent explained that a rule should not be applied, they decided *Never* pay off the technical debt item. They explained that the rule application did not precisely identify a technical debt item, e.g. “This is a stupid rule. Sometimes verifying that the given code executes without throwing is all you need.” They also explained that some rules are wrong based on program language documentation, as in the previous example about Javadoc documentation.

ARBITRARY_RULE: For some developers, some rules should not be applied because they believe that the kind of technical debt found by the rule is not technical debt. Thus, the rule should be ignored - for instance, rules that try to anticipate possible runtime errors.

All developers that indicated an arbitrary rule chose *Never* to pay off the technical debt item, such as in: “This evaluation seems very wrong — how can a static method call throw an NPE?”, and “The rule is wrong, it actually throws NoSuchElementException, but the linter is unable to detect it. It probably has an incomplete type system and flow analysis.”

FALSE_POSITIVE: In some contexts, a technical debt item is considered a false positive. Unlike the rules that should not be applied to a project or the rules that wrongly identify technical debt items, this code refers to cases in which, in another code snippet, the item might be considered technical debt. However, it does not make sense as technical debt for that snippet of code.

Every time a respondent indicated a false positive, they chose *Never* to pay the technical debt. Many used the term "false positive", but others said more, e.g. “Once again, default case may or may not make sense: in cases where it is omitted it is literally useless.”

Super-Category UNUSED_CODE: includes the following categories:

UNUSED_CODE: Code to test a concept or idea, but later, is replaced by some better code or, after writing it, the developer realizes it is not a good concept/idea. Some code is considered useless because it was written as an example to show or teach.

All developers chose *Never* to pay off technical debt where the code was unused. In some cases, the item was inside a test class, or private, or just never used.

Super-Category CODE_IMPROVEMENT includes the following categories:

PERFORMANCE: Code written by a developer is not always the one with the best performance. To improve performance the developer could spend hours or even days to rewrite the code to reduce complexity. They can also simply use some programming language features, such as built-in or call methods that perform processing in parallel, resulting in significant improvement in the performance of the software.

One respondent chose to pay off a technical debt item *As soon as possible* because they saw it specifically to be a performance issue.

REMOVE_BUG: When developers write code, they cannot cannot always see all possible situations. So sometimes there is a need to rewrite the code in order to remove bugs that are generated by the wrong use of logic or because the code is not covering every possible case.

One respondent decided to pay off the technical debt item *As soon as possible* because: “A better exception here would be a good idea. This might even be a bug.”.

TEST_FAIL: Sometimes code can break the tests. This happens because the code was written incorrectly and therefore returns unexpected behavior, in which case the code must be rewritten to behave as the test expects. It may also have been written with a different structure than expected by the tests and therefore the test also fails. In this case the code structure can be changed to adapt to the test or the test can be changed to adapt to the code structure.

One respondent chose to pay off the technical debt *Immediately* because it broke a test: “It should be paid immediately because the test is broken. The expected value is "Number of created files" and the files.size() is an integer.”

IMPROVE_READABILITY: A code needs to be rewritten to be easier to read. Generally, rewriting can be done by renaming variables, classes, methods/functions. It can also be done by decreasing complexity, such as removing "if", "for" and "while" statements. It can also be done through refactoring the code or using methods that encapsulate part of it. These techniques make the code easier for other developers to understand.

A respondent decided to correct one of the identical sub-expressions on both sides of operator "||" C because he thought it was “Confusing”. Another developer decided to use isEmpty() As soon as possible to check whether the collection is empty or not to make it “clearer”.

INCOMPLETE_CODE: The code is incomplete when some implementation is missing for the method/function to work as expected. Most of the time, developers annotate these missing parts with the following comment: "TODO: explanation", where TODO means the code needs to be written, and sometimes there is an explanation about what needs to be written and/or how.

Sometimes this code leads to *Immediate* payment and sometimes the payment should be done *As soon as possible.* A developer chose to pay two technical debt items *Immediately* because he considered that TODO annotation should be paid off urgently: "From my point of view, TODOs should be urgent, otherwise people are going to leave it there and procrastinate as much as they can."

Super-Category COST_BENEFIT includes the following categories:

LOW_PRINCIPAL: The code is easy to modify, that is, in a few minutes a developer is able to fix the problem or complete the logic. Therefore, the technical debt item has a low principal cost.

Respondents tended to pay off these items either *Immediately* or *As soon as possible*. They used terms such as "trivial", "easy and safe", and "easy change". In one case, the respondent thought that the change could even be automated.

HIGH_PRINCIPAL: The code is hard to modify, that is, a developer could take several hours or even days to (re)write the code to fix the problem or complete the logic. Therefore, the technical debt item has a high principal cost.

In one case, a developer thought that the technical debt item should be paid off *As soon as possible*, but "This TODO should be treated with more care because it can break the existing code". Another developer chose *Never* to pay the debt because the effect was minimal and the complexity to remove the item was huge: "This TL works as part of injected code during test runs. The overhead is minimal and the complexity of calling remove is huge."

LOW_INTEREST: The code is almost never called by other methods/functions, that is, it has a low probability of causing extra effort if the item is not paid off. Therefore, the technical debt has a low interest.

One respondent chose to pay off an item *In the next release* because he did not face that issue at that moment: "Since it might define other method interfaces, I would prioritize this issue to the next release. In this code, we do not face this issue at the moment because the method is private and the public methods do not re-throw the exception." The same developer also chose to pay off *In the next release* another technical debt item because it had a minor impact: "Good catch. Minor impact/nit." Another developer chose *Never* to pay the technical debt item because it had a low impact and high effort to pay off: "This TL works as part of injected code during test runs. The overhead is minimal and the complexity of calling remove is huge.". Another developer also chose not to pay off an item because it is inside a private method.

5.2.3 Selective Coding

We performed selective coding to refine and integrate categories. Our main goal was to understand the main criteria developers use to decide whether a technical debt item should be paid off and when to make the payment. We used abstraction to incorporate all aspects related to the collected data and coding Anselm Strauss and Juliet Corbin, 1998.

The codes and categories fell naturally into those representing influences on the decision to pay off technical debt and those influencing the decision *not* to pay off technical

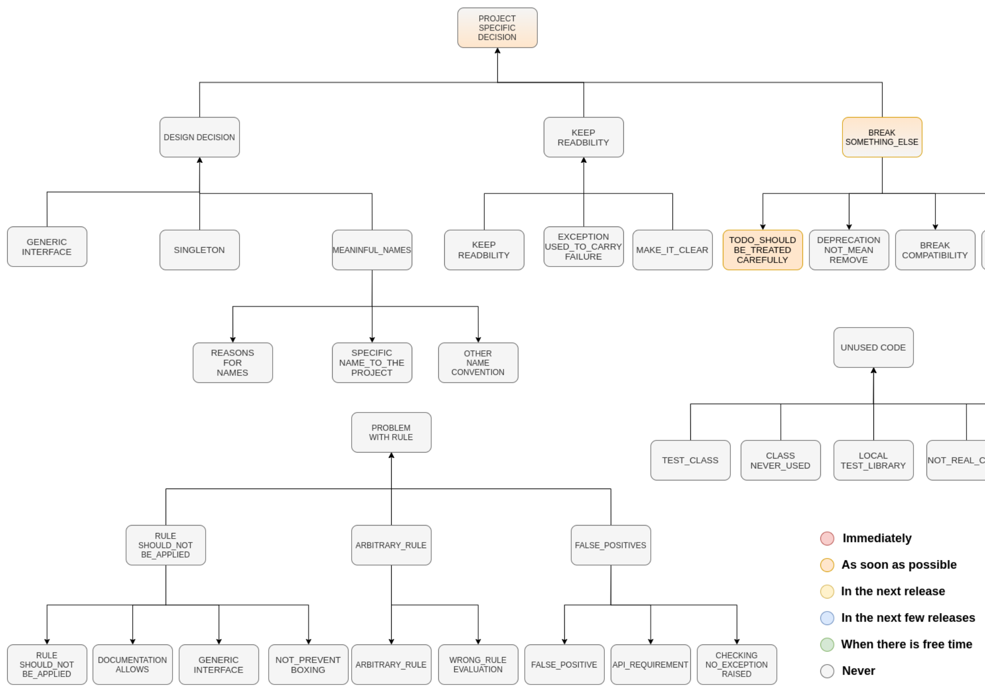


Figure 5.2: *Codes and categories for not paying off technical debt items.*

debt. The super-categories PROJECT_SPECIFIC_DECISION, PROBLEM_WITH_RULE and UNUSED_CODE represent the codes describing decisions not to pay off the technical debt item. Figure 5.2 shows the categories and super-categories related to decisions *not* to pay off the technical debt item.

In some situations, a PROJECT_SPECIFIC_DECISION is made to keep the current solution (i.e. not pay off the debt) instead of rewriting the code to follow the rules. These are project-specific decisions because a similar situation in a different project might lead to a different decision about paying off the debt. Sometimes these decisions are related to architectural design (DESIGN_DECISION); for example, when a generic interface is designed to propagate any checked exception or singletons are used as constants. Another type of project-specific decision is related to naming conventions (MEANINGFUL_NAMES), i.e. when a project prefers to use a different way to name variables, methods, functions, and classes than conventions commonly used. Another type of project-specific decision is related to a concern about readability (KEEP_READABILITY); the decision is not to change the code and follow the rules because that would make the code harder to read. Finally, developers also decided not to pay the technical debt item when it would break compatibility or functionality (BREAK_SOMETHING_ELSE); fixing a code snippet would imply changing a lot of other code snippets that depend on the first one. The only exception is when it breaks something else, but it is essential to pay off a debt (TODO_SHOULD_BE_TREATED_CAREFULLY) as soon as possible; that is when a missing snippet needs to be written to provide the expected behavior.

The PROBLEM_WITH_RULE category describes cases where a respondent cites a problem in a rule used to detect technical debt items in SonarQube. Respondents felt some rules should not be applied (RULE_SHOULD_NOT_BE_APPLIED) because they identified irrelevant technical debt. Others felt some rules were arbitrary (ARBITRARY_RULE), not logical, or incorrectly evaluated a technical debt item. Some cases were cited as false positives (FALSE_POSITIVE); that is, the respondent considered the found item not to be a technical debt item, at least in the specific situation.

Developers sometimes wished not to remove or change code that exists just as an example, even though the code has been superseded by other code or is no longer used. Sometimes, developers want to keep the UNUSED_CODE for future comparison. Some projects are created as proof-of-concept or just to teach software development.

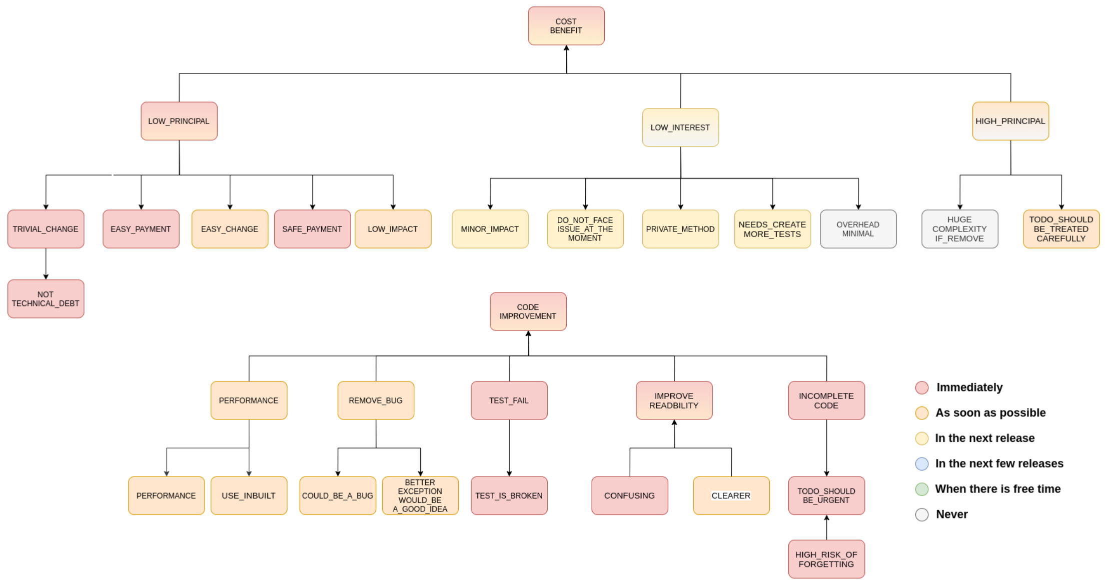


Figure 5.3: *Codes and categories for paying off technical debt items.*

The super-categories CODE_IMPROVEMENT and COST _BENEFIT summarize the influences for developers deciding to pay off the technical debt item, as shown in Figure 5.3.

When a developer's reason for paying off a technical debt item fell into one of the CODE_IMPROVEMENT categories, there are interesting relationships between the specific reason and how quickly they wanted to pay off the debt (i.e. "immediately" or "as soon as possible"). When the reason to pay off the debt is to improve the performance (PERFORMANCE) or remove bugs (REMOVE_BUGS), developers indicated that it should be paid off as soon as possible. On the other hand, they decided to pay off the technical debt immediately when the reason was a failed test (TEST_FAIL) or incomplete code (INCOMPLETE_CODE). Finally, when the motivation is to improve readability (IMPROVE_READABILITY), they want to pay off the debt immediately when the code is confusing, and as soon as possible when they want to clarify the code.

The other super-category describing cases where the technical debt item should be paid off is COST_BENEFIT. There are also variations in how quickly the debt should be paid off in these cases. When the criteria used were low principal (LOW_PRINCIPAL), respondents immediately chose to pay off the technical debt item because they considered it a trivial change, an easy and safe payment. However, when the debt item was considered an easy change or low impact, they decided to pay it off in the next release. When the criteria used was low interest (LOW_INTEREST), respondents chose to pay off the technical debt item in the next release, except when overhead was minimal (OVERHEAD_MINIMAL), and the complexity to remove it was huge. About (HIGH_PRINCIPAL), they decided not to pay it when there was a huge complexity to do so, and to pay off as soon as possible when there is a TODO tag that should be treated carefully.

5.3 Discussion of the Results

The developers used a wide array of criteria to decide whether and when a code technical debt item should be paid off. They had different motivations when deciding to pay off a technical debt item in different situations. We grouped these criteria into three super-categories that define when a technical debt item should not be paid off and two that define when an item should be paid off.

Another interesting observation is that when developers decide to pay a technical debt item off they want to do it soon: immediately, as soon as possible, or in the next release. However, we could not determine through data if this decision was technically based or if personal feelings (worry, anxiety, fear of the reputation) made them choose higher priority actions.

In addition, they used specific criteria for each priority level. This means that when they use the same criterion in different instances, they choose the same priority level or a neighboring one. For instance, when the technical debt item was about improving readability, participants always decided to pay off those debts immediately or as soon as possible, i.e. they chose neighboring categories. The codes OVERHEAD_MINIMAL and HUGE_COMPLEXITY_IF_REMOVE are the exceptions to that observation, as there was more variety in developers' answers related to these criteria.

Another finding is that each software project needs a specific set of rules to identify technical debt items that are relevant to the project. More than half of the answers were never to pay off the technical debt item. For many of these cases, the respondents explained they were using their preferred pattern than the one identified as technical debt by the SonarQube detection rules.

We identified categories similar to the decision-making criteria presented by Riegel and Doerr RIEGEL and DOERR, 2015 and Ribeiro et al. L. RIBEIRO *et al.*, 2016. Like Leppanen et al. LEPPANEN *et al.*, 2015, we present a framework to decide on the payment of a technical debt item, but our model is based on the decision criteria of the respondents. The first and fourth findings presented in this discussion confirm the literature. However, the second and third findings are new completely new.

5.3.1 Implications for Researchers and Practitioners

The results of this study could be used as the basis for researchers to identify other criteria that developers use to decide whether and when to pay off a technical debt item. This study could be replicated in other software project groups to identify new decisions and prioritization criteria, such as applying it to projects that use other programming languages and non-OSS projects. Besides that, other methods, such as interviews, could be used to identify and better understand the criteria.

After defining criteria, researchers could study the scenarios in which they are used, that is, to define when and how each criterion is used based on project variables like a programming language, project patterns, project size, development methodology, development team size, and others. With these definitions, it is possible to create guidelines to assist developers in deciding the payment priority level for each technical debt item they identify in their software. In addition, researchers can relate the criteria to software context, such as code metrics and commit history, to automatically categorize payment of code technical debt items using, for instance, machine learning.

Practitioners can use the criteria we have found to evaluate and plan technical debt payment pragmatically in their projects. That is, they could verify that the criterion applies to the project, and if so, they use it to define its priority level of payment. In addition, practitioners who develop technical debt management tools could use the criteria list identified here to improve the technical debt identification tools. After further research, they could use the criteria to tune technical debt management software for each project established context based on the guidelines.

5.3.2 Threats to Validity

In this study, we adopted a careful approach to mitigate possible biases and misinterpretations. Below, we describe the steps we took to reduce threats to validity.

Construct validity: We conducted a pilot study by applying the questionnaire to fifteen developers. First, they answered the survey about their projects, then they answered questions about their experience using the survey tool and suggested improvements. Based on the review answers, we fixed some issues in the tool and applied improvements to ensure the data items reflected a consistent interpretation of the study constructs.

External validity: This study can be replicated in other software projects. For that, analyzing the project with SonarQube and using the survey tool to identify technical debt items and collect answers will be necessary. We have collected answers from 21 open-source Java projects with many sizes and features. Future replication should be conducted on more projects, other programming languages, and non-OSS projects. Applying an interview can also help understand the prioritization criteria and make them more general.

Internal validity: We analyzed the answers one by one and applied codes to them. After that, we tried to improve and interpret the codes through grouping. Then, we reviewed them several times to ensure they accurately represented the answers. We extracted the conclusions and verified that they all were derived from the data. We followed the standard guidelines for qualitative coding reviewed by other authors. The iterative approach helps to mitigate analysis biases. In addition, we specifically designed the survey to capture the relationship between criteria and decisions to pay off or not technical debt items. Thus the conclusions about that relationship come directly from the data.

Reliability: We followed Straussian GT analysis techniques to interpret the data. Initially, one author conducted the coding and analysis. Then, the other authors revised, discussed, and iteratively improved the codes and analysis. We carefully documented all steps.

5.4 Conclusions and Future Work

In this study, we performed a survey to collect data on open-source Java software projects hosted on GitHub to understand which criteria software developers use in practice to decide whether and when code technical debt items should be paid off. We asked the participants questions about technical debt items from the projects they had contributed to.

We analyzed the data using Straussian GT techniques to identify developers' criteria for prioritizing technical debt. We grouped the criteria into 15 categories. Then, we grouped them into two super-categories related to paying off the technical debt item; and three super-categories related to not paying off the item.

We observed that some participants decided not to pay off a technical debt that had occurred because of a specific project decision. However, when participants decide to pay off an item, they want to do it soon. Another observation is that all respondents who use a particular criterion choose the same priority level or a neighboring one. Finally, we noted that each software project needs specific rules to identify its technical debt.

In future work, it will be essential to expand the number of projects and participants to cover more kinds of projects (industrial projects, different programming languages, for instance) and types of technical debt (architectural debt, for instance).

Chapter 6

Technical Debt Prioritization using Machine Learning

In this chapter, we aim to prioritize the payment of technical debt items in real software projects in an automated way. To achieve that goal, we applied a machine learning method approach to decide whether a TDI should or not be paid off and when the payment should be made. Using an iterative survey, we collected 2,616 answers from 276 developers over 207 Java public software projects hosted on GitHub regarding payment priority technical debt items.

We used Sonarlizer Xplorer 4.1 tool to find the software projects repositories, analyze them to find technical debt items, and relate the projects to their respective developers. We extracted 27 code metrics from the project's source code. We used the metrics as methods features to apply to the supervised ML methods to prioritize technical debt items. Finally, we performed an experiment to evaluate the effectiveness of using ML to prioritize technical debt items.

The research question for this study is:

- RQ2. How to prioritize the payment of technical debt automatically?

We split the main question into three subquestions:

- RQ2.1. How effective are Machine Learning models for deciding **whether** or not a technical debt item should be paid?
- RQ2.2. How effective are Machine Learning models for deciding **when** a technical debt item should be paid?
- RQ2.3. Which are the best Machine Learning algorithms to prioritize technical debt?

6.1 Methodology

In this study, we developed and evaluated context-adaptive machine learning methods to assist in technical debt prioritization. First, we used the Sonarlizer Xplorer (Pina, Goldman, and Seaman, 2022) to build a dataset with code technical debt items from Java projects hosted on public GitHub repositories. We used InterasurveyTD (Pina, Goldman, and Seaman, 2022) to apply a survey to the project developers. The survey asked developers about the payment priority level for technical debt items for software projects they have contributed. Finally, we trained and tested well-known supervised machine learning methods to decide whether and when a technical debt item should be paid off.

6.1.1 Data Collection

The data collection process comprised four main steps: mining Java software projects hosted on public GitHub repositories related to the contributed projects and files, identifying technical debt items and code metrics, and applying a survey to developers about technical debt prioritization. Figure 6.1 shows the data collection process.

Project Selection and Independent Variables

In our study, we employed the Sonarlizer Xplorer tool 4 to conduct data mining on public Java software repositories and developers sourced from GitHub. This tool identifies code technical debt items and extracts code metrics using SonarQube. Furthermore, it enabled the establishment of associations between developers and the code files they contributed to.

In addition, we used the InteraSurveyTD tool 4 to conduct a survey in which developers responded to questions regarding prioritizing technical debt items for projects they actively contributed to.

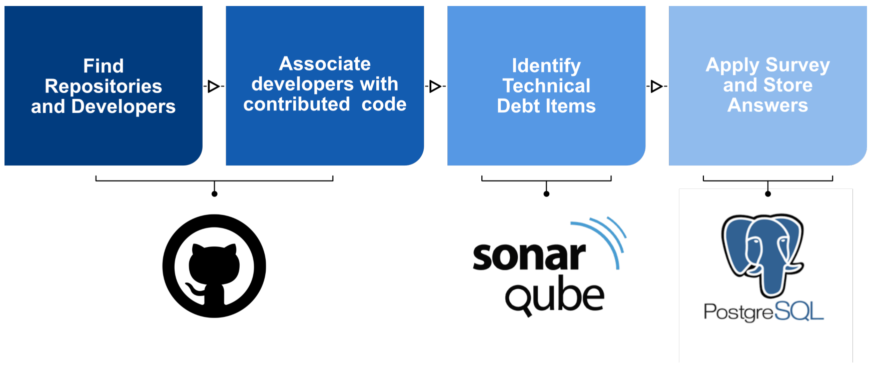

Find Repositories and Developers
Associate developers with contributed code
Identify Technical Debt Items
Apply Survey and Store Answers
sonar qube
PostgreSQL

Figure 6.1: *Data collection process.*

Our analysis encompassed 45,944 Java software projects, openly available on GitHub repositories. We identified 15,217,381 technical debt items using SonarQube. We sent an email invitation to 145,091 developers to answer our survey, of which 276 developers of 207 projects provided answers for 2,616 technical debt items.

The selected projects are diverse with respect to the number of non-commented lines of code (ncloc), number of developers, amount and type of technical debt, and complexity. For example, Figure 6.2 shows the distribution of complexity (on the top) and ncloc (on the bottom) for selected projects for which we have at least one developer response. Our set includes Nginx Java Parser, Fast Login, and SQLite JDBC projects.

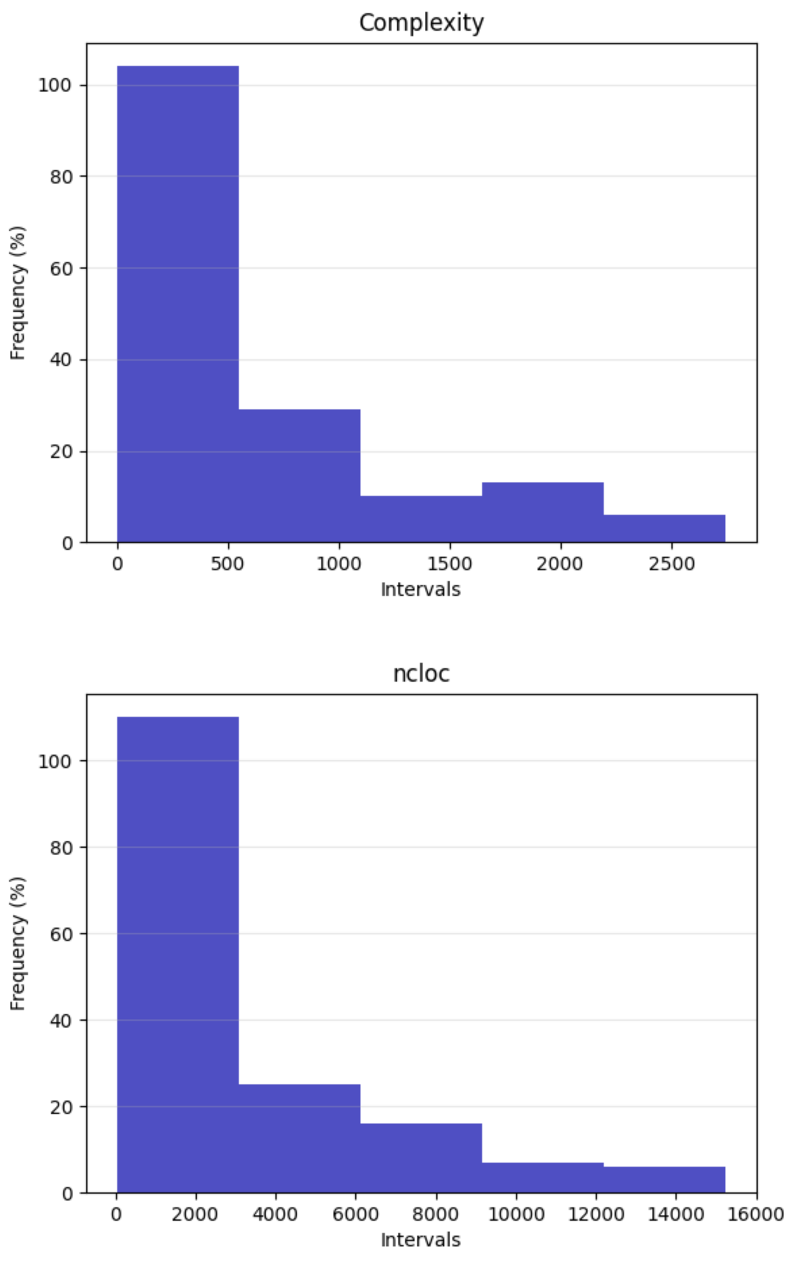


Figure 6.2: *Complexity and ncloc distribution for selected projects where a developer answered.*

SonarQube uses quality rules to identify technical debt items. When a rule is violated, it creates an item. In this study, the "technical debt type" refers to the quality rule that generated the technical debt item. Our dataset has 31 different technical debt types, all related to code technical debt. Below are some examples of the types of technical debt:

- File names should comply with a naming convention;
- Child class methods named for parent class methods should be overrides;
- Cognitive Complexity of methods should not be too high.

We used 27 code metrics extracted from SonarQube as features (independent variables) to train the machine learning models. Table 6.1 shows the selected code metrics used as features, short descriptions, and statistics.

Our final dataset[1] used for training and assessment was composed of columns for these 27 metrics and 1 column containing the developer response (the dependent variables). It also includes other columns extracted from SonarQube for administrative purposes (e.g. filename, project name), that are unsuitable for use in machine learning algorithms but may be useful to future researchers. This dataset has 2,616 rows, one for each survey answer.

Survey and Dependent Variable

We used the InteraSurveyTD tool 4 to apply an interactive survey to the software project developers showing technical debt items from projects they contributed and asking about technical debt prioritization. We sent emails [2] to invite the developers containing a project brief and a link to the survey with an identifier to load the technical debt items related to that developer. Once he followed the link, they saw instructions for the survey and informed consent document[3], approved by the UMBC Institutional Review Board. They had to confirm having read and agreed with consent terms and to be at least 18 years old before starting the survey. Then, the survey tool chose a random technical debt item to show the developer.

InteraSurveyTD doesn't show all the items from the developer's files to preserve anonymity. It randomly picked up a technical debt item for the developer only 70% of the time. This technique avoids identifying the developers in files with only one contributor. It also didn't store the relationship between the answer and the developer. Finally, all selected projects have at least three participants to ensure anonymity and prevent the participants' answers from being tracked.

The survey technical debt prioritization question had six answer options on a descending scale according to how urgently the item should be paid off:

- **Immediately**: pay the item off before developing anything else;
- **As soon as possible**: pay item off in the current release;
- **In the next release**: plan item payment for next release;
- **In the next few releases**: it doesn't postpone payment indefinitely, but it doesn't have to happen in the next release;
- **When there is free time**: no planning is required, but eventually the item should be paid;
- **Never**: the item is not important for the project, or it is not in fact, a technical debt item, or for some other reason, should not be paid.

[1] https://zenodo.org/record/7709535/files/dataset.csv

[2] https://zenodo.org/record/7709535/files/email-template-anonymized.pdf

[3] https://zenodo.org/record/7709535/files/research_web_consent-final.pdf

Metric	Description	M	SD	Q1	Mdn	Q3	Skew
tdtype	Technical debt type.	-	-	-	-	-	-
ncloc	Number of non commented lines of code of the TDI file.	350	781	41	106	342	5.3
lines	Number of lines of the TDI file.	494	1081	70	148	475	5.4
classes	Number of classes in the TDI file.	2	14	1	1	1	24.5
functions	Number of functions in the TDI file.	26.7	93.4	3	7	20	8.8
statements	Number of statements in TDI the file.	146.7	362.1	10	38	138	7.1
comment_lines	Number of lines commented.	55.4	193.9	1	6	33	7.9
comment_lines _density	Number of lines commented over number of lines in the TDI file.	10.1	12.6	0	5	15	2.0
complexity	Cyclomatic complexity in the TDI class.	75.0	219.4	5	17	60	9.3
complexity_file	Cyclomatic complexity in the TDI file.	75.0	219.4	5	17	60	9.3
cognitive _complexity	Cognitive complexity in the TDI file.	71.8	257.9	1	11	46	10.2
duplicated_lines	Number of duplicate lines in the TDI file.	18.3	97.5	0	0	0	7.9
duplicated_blocks	Number of duplicate lines blocks in the TDI file.	1.2	9.8	0	0	0	17.0
duplicated_lines _density	Number of duplicate line over number of lines in the TDI file.	3.2	11.2	0	0	0	4.5
violations	Number of TDIs in the TDI file.	133.5	829.6	3	8	27	8.4
blocker_violations	Number of TDIs classified as blocker in the TDI file.	0.5	1.8	0	0	0	8.3
critical_violations	Number of TDIs classified as critical in the TDI file.	9.6	73.6	0	0	3	14.2
major_violations	Number of TDIs classified as major in the TDI file.	106.2	826.0	1	2	7	8.6
minor_violations	Number of TDIs classified as minor in the TDI file.	16.6	52.1	0	2	8	6.7
code_smells	Number of code smells in the TDI file.	131.0	829.5	3	7	24	8.4
bugs	Number of bugs in the TDI file.	0.8	4.2	0	0	1	21.9
sqale_index	Total effort in hours to fix all the issues on the TDI file.	1234.3	8163.7	20	60	216	8.4
sqale_rating	1-to-5 rating based on the technical debt ratio.	1	0	1	1	1	3.2
sqale_ratio	Ratio of the actual technical debt compared to the estimated cost to develop the whole source code from scratch.	3.5	8.8	0	1	3	6.1
reliability_rating	Reliability rating of the TDI file.	1	1	1	1	2	1.8
security_rating	Security rating of the TDI file.	1	0	1	1	1	3.5
security_review _rating	Security review of the TDI file.	1	1	1	1	1	2.2

Table 6.1: *Selected code metrics*

Figure 6.3 shows the statics about survey invitations, access, and answers.

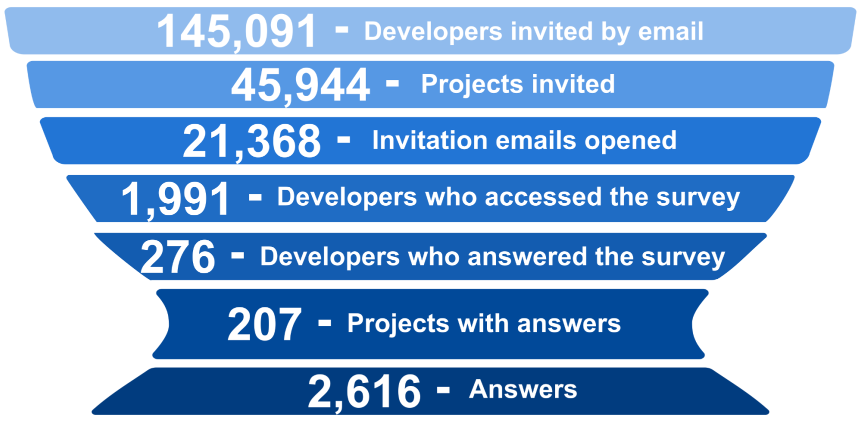


Figure 6.3: *Survey Process Flow Statistics*

6.1.2 Data Preparation

Table 6.2 shows the answers labels and count. Once the data was collected, we categorized the survey responses into three distinct label types: "priority," which directly associates the answer with a specific label; "3-classes," consisting of high, medium, and low labels indicating when the payment should occur; and "pay or not," represented by yes or no labels, determining whether a payment should be made. The distribution of answers for each label type can be observed in Figure 6.4.

Priority	3-classes	Pay?	Answer	Count
1	(1) High	(1) Yes	Immediately	240
2	(1) High	(1) Yes	As soon as possible	272
3	(2) Medium	(1) Yes	In the next release	318
4	(2) Medium	(1) Yes	In the next few releases	227
5	(3) Low	(1) Yes	When there is free time	698
6	(3) Low	(0) No	Never	861
Total				2,616

Table 6.2: *Answer labels and count*

In order to create the "pay or not" label we set the label 0 (No) when the answer is 6 (Never). Otherwise, it means it should be paid off; we set the label to 1 (Yes).

In addition, we grouped the answers into three priority categories: (1) high, with the answers "Immediately" and "As soon as possible"; (2) medium, with "In the next release" and "In the next few releases"; and (3) low, with "When there is free time" and "Never".

In addition, we grouped the answers into three priority categories:

1. **High**, with the answers "Immediately" and "As soon as possible";
2. **Medium**, with "In the next release" and "In the next few releases";

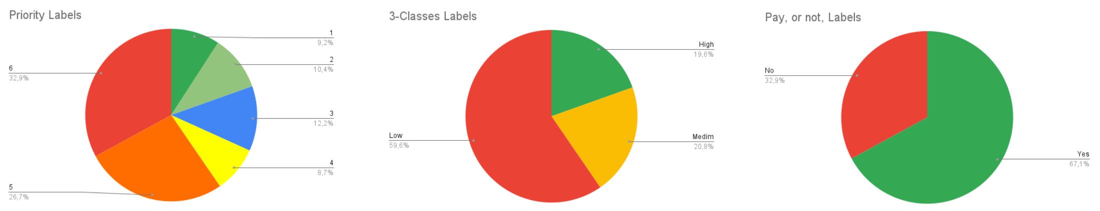


Figure 6.4: *Answers distributions by Label*

3. **Low**, with "When there is free time" and "Never".

After investigating manually several cases related to empty values for some metrics, we figured out that when the value is empty, no occurrence of that metric was found. For example, the number of critical violations or complexity. For this reason, we set the empty values to 0.

We intentionally chose very diverse software projects in terms of the types of projects and their context. Therefore, we did not expect the independent variables to follow normal distributions. For the same reason, we decided not to withdraw outliers to avoid removing the diversity of contexts we purposely included.

6.1.3 Exploratory Analysis

We need to verify the existence of a relationship between each of the selected metrics and the dependent variable to use the metrics as independent variables in our machine learning classification models. In this study, we worked with three dependent variables (priority, 3-classes, and pay or not label). Thus, we had to investigate the relationship between each label's selected metrics. For this reason, our exploratory analysis was done by studying the discriminative power of the chosen metrics using hypothesis testing.

We tested the null hypothesis that the distributions of each metric for each label class are equal to perform the exploratory analysis and hence to determine whether the 27 selected metrics can discriminate between label classes. For example, for the 3-classes label, the null hypothesis will verify whether the number of non-commented line metric (ncloc) distributions is equivalent to high, medium, or low distribution.

We applied a Shapiro-Wilk (Shapiro and Wilk, 1965; Razali, Wah, *et al.*, 2011) test for normality, resulting in a normal distribution for each metric distribution. For this reason, we used the non-parametric Mann-Whitney U test and tested our hypothesis at a 95% confidence level ($\alpha = 0.05$). For each dependent variable and all selected metrics, the p-value from Mann-Whitney U test (Mann and Whitney, 1947) was less than $\alpha = 0.05$. Thus, in all tests, we rejected the null hypothesis. The test suggested a statistically significant difference between each metric and the dependent variable. Thus, no metric describes the dependent variable completely. Therefore, all of these metrics can discriminate and potentially be used as predictors of technical debt prioritization. Table 6.3 shows the discriminative power analysis result for the Mann-Whitney U test.

Metrics	Mann-Whitney U test (p-value)		
	Priority	3-class	Pay or Not
rule_id	0.0	0.0	0.0
ncloc	0.0	0.0	0.0
lines	0.0	0.0	0.0
classes	0.0	0.0	1.538e-165
functions	7.177e-80	3.421e-230	0.0
statements	0.0	0.0	0.0
complexity	1.676e-253	0.0	0.0
file_complexity	1.676e-253	0.0	0.0
cognitive_complexity	4.452e-55	3.837e-108	4.879e-296
comment_lines	1.573e-22	2.586e-81	1.640e-290
comment_lines_density	3.157e-10	5.751e-51	1.278e-233
duplicated_lines	0.0	0.0	7.5837e-203
duplicated_blocks	0.0	0.0	2.693e-236
duplicated_lines_density	0.0	0.0	3.969e-213
violations	1.087e-133	0.0	0.0
blocker_violations	0.0	0.0	3.941e-232
critical_violations	0.0	7.392e-200	0.0001
major_violations	1.593e-38	0.440	9.932e-282
minor_violations	4.355e-47	8.979e-06	1.235e-163
bugs	0.0	0.0	1.101e-120
code_smells	4.279e-99	1.981e-260	0.0
sqale_index	0.0	0.0	0.0
sqale_debt_ratio	1.0127e-289	2.447e-150	8.456e-92
sqale_rating	0.0	0.0	8.204e-294
reliability_rating	0.0	0.0	0.0
security_rating	0.0	0.0	0.0
security_review_rating	0.0	0.0	7.758e-277

Table 6.3: *Discriminative power analysis using Mann-Whitney U test*

6.1.4 Model Building

The process of building machine learning models to predict a technical debt payment item involves three key steps: model selection (6.1.4), testing methods and parameters to identify optimal configurations for prediction in different contexts (6.1.4), and evaluating the performance of the models to address the specific research questions (6.1.4). These sequential steps ensure the systematic development and assessment of the machine learning approaches for accurate prediction of payment prioritization outcomes.

Model Selection

The data collection process takes time and resources. Even with tens of thousands of projects analyzed and emails sent, it resulted in only over two thousand survey answers. Therefore, we initially rejected deep learning and other techniques requiring a large data volume.

We decided to use machine learning methods for technical debt item payment prioritization because they have not been previously explored. We also decided to select well-known supervised machine learning methods that have already been used to solve similar problems in software engineering, e.g., identifying TD (Tsoukalas, Mittas, *et al.*, 2021), predicting refactoring (Mauricio Aniche *et al.*, 2020), and determining bug severity (Chaturvedi and V. Singh, 2012). The hyperparameters were selected based on previous work and references in the literature (Yang and Shami, 2020; Tsoukalas, Mittas, *et al.*, 2021; Mauricio Aniche *et al.*, 2020).

Table 6.4 describes the nine selected machine learning methods and the hyperparameter values we tested to tune the algorithm.

We can infer from Figure 6.4 and Table 6.2, the answer distributions are imbalanced for all three label types. That is, the number of instances for each answer class is not the same. When applying machine learning methods to imbalanced data, there is a risk of generating biased results due to the algorithms favoring the majority class. In the case of the "pay or not" label, selecting the majority pay class would yield an average accuracy of 67%, whereas a random choice would result in 50% accuracy. To mitigate this issue, we employed a technique called oversampling (Mohammed *et al.*, 2020) to address the class imbalance. Unlike undersampling, which discards data, oversampling randomly duplicates instances of the minority classes until they are equal to the majority class. This approach ensures a more balanced class distribution and helps prevent data loss, thereby improving the reliability of the machine learning analysis (Mohammed *et al.*, 2020).

Model Testing

We performed the training-validation-test approach to evaluate the methods and verify which ones are the best for each scenario. We randomly partitioned the dataset into two sets: training/validation with 80% of the data and testing with 20%. We used the first partition to train the machine learning methods varying the hyperparameters to find and validate the combination that maximizes accuracy and F1 (as a tier). Then, we retrained the methods using the training/validation dataset and the hyperparameters, resulting in the highest accuracy to assess their performance on the test dataset.

Method Name	Description	Hiperparameter values
Dummy Classifier (DC)	It serves as a baseline for comparison, its behaviors are based on strategy parameters.	**strategy**: most_frequent, prior, stratified, uniform, constant; **constants**: 0, 1; **random_state**: $[0, 100]$
Gaussian Naive Bayer (NB)	It uses Bayes' theorem to calculate the probability of each class based on the data from feature values.	**var_smoothing**: $[0, 10^{-9}]$
K-Nearest Neighbors (KNN)	It classifies based on the distances between the data points.	**n_neighbors**: $[1, 20]$, **weights**: uniform, distance; **algorithm**: auto, ball_tree, kd_tree, brute; **p**: $[1, 40]$
Logistic Regression (LR)	It is a linear model that implements a logic function to predict the probability of a target class belonging to a certain class.	**solver**: newton-cg, lbfgs, liblinear, sag, saga; **penalty**: l1, l2, elasticnet, none; **C**: $[0.01, 100]$
Ridge Classifier (RC)	It converts the target values into $[-1, 1]$ and then treats the problem as a regression.	**alpha**: $[0, 10]$; **solver**: auto, svd, cholesky, lsqr, sparse_cg, sag, saga, lbfgs
Support Vector Machine (SVM)	It looks for the best hyper-plane in high-dimensional space to separate the data into classes.	**C**: $[0.01, 10]$; **kernel** linear, poly, rbf, sigmoid
Decision Tree (DT)	It uses a tree-structure model to define a set of classification rules from the data.	**criterion**: gini, entropy, log_loss; **max_depth**: $[5, 50]$; **min_samples_split**: $[2, 11]$; **min_samples_leaf**: $[1, 11]$; **max_features**: auto, sqrt, log2; **spliter**: best, random; **min_weight_fraction_leaf**: $[0, 1, 2, 4, 8]$; **max_leaf_nodes**: $[None, 20, 100, 500, 1000]$
Random Forest (RF)	It fits several DT classifiers on many sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting.	All DT values + **n_estimators**: $[10, 100]$
XGBoost (XGB)	It is based on multiple DTs and uses gradient boosting to minimize loss.	**objective**: binary:logistic; **booster**: gbtree, gblinear, dart; **n_estimators**: $[10, 200]$; **max_depth**: $[5, 50]$; **learning_rate**: $[0.01, 1]$; **subsample**: $[0.01, 1]$; **colsample_bytree**: $[0.01, 1]$

Table 6.4: *Selected Machine Learning methods, description, and hyperparameter values*

Before starting the training/validation phase, we apply oversampling to the training dataset statistically maintain the data balance among labels. Then, we performed a stratified 5-fold cross-validation (Refaeilzadeh *et al.*, 2009). It randomly divides the dataset into five folds. This involved partitioning the dataset into five subsets or folds, where each fold was used as the test set while the remaining four folds were used for training. The training and validation process was repeated for each of the five folds, ensuring that every subset served as both training and test data. During this iterative process, we computed the mean accuracy and F1 score as performance metrics to compare the effectiveness of different hyperparameters and machine learning methods. These measures allowed us to assess and compare the overall predictive performance across the various configurations and methods.

We also vary the hyperparameters to find the parameter values that increase the predictive power for each machine learning method in each case. We based our experiment on prior work (Yang and Shami, 2020; Tsoukalas, Mittas, *et al.*, 2021; Mauricio Aniche *et al.*, 2020) to select which parameters to vary and which values to use in our tuning process. However, some methods could take months to combine parameters and run tests for many values. Therefore, we chose to use Random Search (Yang and Shami, 2020), which randomly picks a predefined number of combinations of hyperparameter values to evaluate. We defined the search space size at $\#combinations * 1.3$, if $\#combinations < 2^{10}$; $min(\#combinations * 0.5, 2^{12})$, otherwise. We tried to set a large enough search space to detect the global optimal solution, or at least their approximations (Bergstra and Bengio, 2012). To validate the optimal value, we used accuracy as a performance measure, because, in our study context, false positives and false negatives are equally bad for technical debt prioritization. That is, paying for a non-priority item wastes time unnecessarily, and not paying for a priority item affects code quality and may take longer to develop new features or maintain the code. As a tiebreaker, we used the F1 measure, the harmonic mean between precision and recall.

The last step during the training/validation phase was a Min-Max (Jayalakshmi and Santhakumaran, 2011) data transformation by scaling and translating each feature value individually to the [0, 1] range. The purpose of Min-Max normalization is to address the issue of features measured at different scales, which can lead to an unequal contribution to model fitting and potential bias. This normalization technique aims to prevent larger magnitude features from dominating others. However, our study found that Min-Max normalization was not efficient in most cases we tested. It did not yield significant improvements and, in some cases, even led to losses in performance. Due to these observations, we decided not to utilize Min-Max normalization in our study.

We repeated the training/validation process five times to avoid bias related to the selected data. During the test phase, we retrained each classifier five times using the training/validation dataset using fine-tuned hyperparameters, resulting in the highest accuracy for each approach. Then, we applied the trained methods to the test dataset. As the classifiers are applied to data they have never touched before, this step simulates the use of machine learning methods in a real case. During this phase, the methods' performance was assessed in different aspects (see Section 6.1.4) so that we can analyze which ones can be used to prioritize technical debt.

Performance Assessment

For training/validation purposes, we chose the machine learning method that achieved the highest accuracy (using F1 as a tiebreaker) for each approach. Then, we computed and evaluated the main traditional performance assessment metrics for the chosen method: accuracy, precision, recall, and F1-score. All these metrics are essential to a complete evaluation of the methods because we are interested in how well the model predicts the correct answers (accuracy), how well it correctly indicates the class of interest (precision), and how well it finds a class of interest (recall). F1-score is a harmonic mean between precision and recall providing a value that reflects their combination.

We computed the performance metrics using traditional formulas when we were predicting between two outcome labels (pay or not). For example, accuracy is the ratio of the number of correct predictions divided by the number of all predictions. The formula to calculate the accuracy of a machine learning method among n prediction values is:

$$hitA(pred, expect) = \begin{cases} 1 & \text{if } pred = expect \\ 0 & \text{if } pred \neq expect \end{cases}$$

$$accuracy(pred, expect) = \frac{\sum_{i=1}^{n} hitA(pred_i, expect_i)}{n}$$

The priority and 3-classes labels are scalars. That is, as the label number increases, the payment priority decreases. Thus, the smaller the distance between the expected and predicted label, the better the method's accuracy. Therefore, we can write the hit function as follows:

$$hitA(pred, expect) = 1 - \frac{|pred - expect|}{number\_of\_label\_types - 1}$$

$$hitA(1,1) = 1 - \frac{|1-1|}{6-1} = 1 - \frac{0}{5} = 1 - 0 = 1$$

But when, for example, the predicted label is 2 (As soon as possible) and the expected label is 1 (Immediately), the hit function returns:

$$hitA(2,1) = 1 - \frac{|2-1|}{6-1} = 1 - \frac{1}{5} = 1 - 0.2 = 0.8$$

On the other hand, when the predicted label is 6 (Never) and the expected label is 1 (Immediately), the hit function returns its minimum:

$$hitA(6,1) = 1 - \frac{|6-1|}{6-1} = 1 - \frac{5}{5} = 1 - 1 = 0$$

A similar process can be applied to produce tuned precision and tuned recall values.

Precision is a metric that measures the proportion of correctly predicted positive instances out of the total instances predicted as positive. Similar to accuracy, we can calculate a distance between the predicted and expected values. However, when dealing with multiple labels, we need to compute the mean precision for each label. We consider all the rows where the predicted value matches the label to calculate precision for a specific label. The function to compute precision is:

```python
def precision_tuned(target_expected, target_predicted, labels):
   hit_sum = 0
   for labelIdx in range(len(labels)):
      class_label = labels[labelIdx]
      hit_count = hit = 0
      for i in range(len(target_expected)):
         if (target_predicted[i] == class_label):
            diff = abs(target_expected[i] - target_predicted[i])
            penalty = diff / (class_labels_size - 1)
            hit = hit + (1 - penalty
            hit_count = hit_count + 1

      if (hit_count > 0):
         precision = hit / hit_count
         hit_sum = hit_sum + precision
   return hit_sum / len(labels)
```

The recall is the ratio of correctly predicted when the expected is positive. Such as precision, we can compute a distance between predicted and expected. We also can calculate recall as the mean of recalls for each label. The function to compute recall is:

```python
def recall_tuned(target_expected, target_predicted, labels):
   hit_sum = 0
   for labelIdx in range(len(labels)):
      class_label = labels[labelIdx]
      hit_count = hit = 0
      for i in range(len(target_expected)):
         if (target_expected[i] == class_label):
            diff = abs(target_expected[i] - target_predicted[i])
            penalty = diff / (class_labels_size - 1)
            hit = hit + (1 - penalty
            hit_count = hit_count + 1

      if (hit_count > 0):
         precision = hit / hit_count
         hit_sum = hit_sum + precision
   return hit_sum / len(labels)
```

Finally, F1 is automatically changed with the idea of the distance between predicted and expected labels because it is a harmonic mean between precision and recall. F1 formula is

$$F1 = \frac{2 * precision * recall}{precision + recall}$$

We call this way of calculating considering the distance between what was predicted and what was expected as tuned accuracy, tuned precision, tuned recall, and tuned F1.

We used traditional metrics for all approaches we tried to prioritize TDI payment. We also applied the tuned analysis metrics for cases with more than two labels because if there are only two labels, the traditional and the tuned analysis result in the same values.

In addition, we employed the Scott-Knott algorithm Jelihovschi *et al.*, 2014 to evaluate the performance of different machine learning methods for each approach. The Scott-Knott algorithm is a hierarchical clustering technique used to identify distinct and homogeneous groups based on the means of the evaluated measures. It is advantageous in scenarios where a significant F-test indicates notable differences between the groups. Our study applied the Scott-Knott algorithm to assess performance measures such as accuracy, precision, recall, and F1 score. We utilized this algorithm to address potential gaps and discrepancies among these measures, ensuring a balanced evaluation. The Scott-Knott clusters were ranked in descending order from A to subsequent clusters (B, C, and so on), indicating that methods within cluster A demonstrated statistically superior performance compared to those in cluster B and subsequent clusters.

6.2 Results

We will answer each of this study's research questions in the following subsections. For each scenario, we selected the hyperparameters that achieved the highest accuracy/F1-score in the training/validation phase. Then, we used the parameter values to retrain and assess the machine learning methods' performance in the test phase. In this last step, we calculated accuracy, precision, recall, F1, and Scott-Knott (SK).

All tables presented in the following subsections have seven columns: machine learning method code (Method), accuracy mean for the training/validation phase (ACC T/V), accuracy, precision, recall, F1-score (F1), and Scott-Knott for the test phase. Accuracy, precision, and recall are the mean values collected in the repetitions.

6.2.1 RQ2.1: How effective are ML models for deciding whether or not a technical debt item should be paid?

We trained the machine learning methods using the "pay or not" label as the dependent variable to indicate whether a technical debt item should or not be paid off. Table 6.5 shows the performance results using traditional analysis.

We can see from Table 6.5 that NB, KNN, DT, RF, and XGB (Cluster A) are the best machine learning methods for deciding whether or not a technical debt item should be paid off, followed by DC, LR, RC, and SVM (Cluster B). In Cluster A, accuracy ranged between 0.69 and 0.88, and F1 ranged between 0.80 and 0.84.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.523	0.516	0.6769	0.519	0.5873	B
NB	0.5604	0.6927	0.6866	0.9885	0.8104	A
KNN	0.854	0.8004	0.8558	0.8414	0.8485	A
LR	0.5951	0.6221	0.818	0.7098	0.7138	B
RC	0.5868	0.6302	0.7049	0.7621	0.7324	B
SVM	0.5847	0.5363	0.7419	0.4661	0.5681	B
DT	0.8499	0.7919	0.867	0.8247	0.8404	A
RF	0.8715	0.8267	0.8533	0.8925	0.8724	A
XGB	0.8829	0.8393	0.854	0.9114	0.8831	A

Table 6.5: *To pay or not to pay machine learning performance*

6.2.2 RQ2.2: How effective are ML models for deciding when a technical debt item should be paid?

This section shows the results of applying four approaches to prioritize when a technical debt item should be paid off. They are 3-classes, the most common TD types, simple, and two-layers.

6.2.3 3-Classes Approach

Our initial approach employed machine learning methods using the 3-classes label to determine when to pay off a technical debt item. This involved categorizing the outcome values, represented by survey responses indicating the preferred payment timing, into three groups: high, medium, and low (as described in Section 6.1.2). By grouping these values, we aimed to enhance the performance of the models by increasing the data volume available for each class defined by the high/medium/low labels.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.3411	0.3302	0.3377	0.3355	0.3054	C
NB	0.3746	0.2786	0.4632	0.3848	0.2618	C
KNN	0.8542	0.6679	0.6145	0.6121	0.6136	A
LR	0.4833	0.4637	0.4387	0.4604	0.4256	B
RC	0.4623	0.4446	0.4267	0.4434	0.41	B
SVM	0.4561	0.3898	0.3892	0.3888	0.3692	C
DT	0.8483	0.6855	0.636	0.6371	0.6349	A
RF	0.8686	0.721	0.6816	0.6533	0.6645	A
XGB	0.8858	0.7229	0.6856	0.6493	0.663	A

Table 6.6: *3-classes approach performance*

The performance outcomes of the 3-classes approach using traditional analysis are displayed in Table 6.6. We can observe that methods within Cluster A exhibit an accuracy nearly twice higher than that of Cluster C, including the method that makes "random" choices. Within Cluster A, accuracy ranges from 0.67 to 0.72.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.8667	0.8401	0.2001	0.6666	0.3078	A
NB	0.8608	0.8464	0.656	0.6716	0.6622	A
KNN	0.9597	0.9099	0.7445	0.7453	0.7449	A
LR	0.8628	0.843	0.6195	0.6351	0.6272	A
RC	0.8531	0.8287	0.5958	0.6068	0.6001	A
SVM	0.8525	0.8194	0.5703	0.5781	0.5742	A
DT	0.9593	0.9115	0.7494	0.7594	0.7543	A
RF	0.9618	0.9201	0.7794	0.7586	0.7688	A
XGB	0.9691	0.9217	0.783	0.7601	0.7714	A

Table 6.7: *3-classes approach tuned performance*

The tuned performance results, as described in Section 6.1.4, are presented in Table 6.7. The results are organized in only one cluster (Cluster A).

Although the difference between the mean of measures of DC (the worst) and XGB (the best) is 0.25, they are in the same cluster their accuracy and recalls are close; in addition, the other methods' measures mean are increasing closely; thus Scott-Knott cannot distinguish between then. Even if the DC precision is low, the other measures do compensation to add it to Cluster A.

Most Common TD Types Approach

In the second approach, we adopted a filtering strategy that focused on the most common technical debt item types instead of applying the selected machine learning methods to the entire dataset. Specifically, we removed rows associated with technical debt types that appeared fewer than ten times. The rationale behind this filtering was to retain only the types with sufficient occurrences. By doing so, the machine learning methods could provide more cases for each value of this independent variable, enabling better learning and potentially leading to improved performance.

In this approach, we employed machine learning methods across all six priority labels in the data. We did not combine the labels into classes but treated them as individual entities. By considering the full range of priority labels, we aimed to capture the variations in the prioritization of technical debt items, thereby allowing the machine learning methods to understand better and differentiate the different levels of priority associated with the items.

Table 6.8 shows the performance results of the most common TD types approach using traditional analysis. The results show that the best methods to prioritize when the payment of a TDI should be made in the TD types that appeared the most in the dataset are KNN, DT, RF, and XGB (Cluster A), followed by LR, RC, and SVM (Cluster B), then DC and NB (Cluster C). The results show that the performance of Class A methods is more than three times better than Class C's "random" method. In Cluster A, accuracy ranged between 0.55 and 0.60, and F1 ranged between 0.50 and 0.55. For Cluster A, the accuracy, precision, and recall are close.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.1768	0.1876	0.1877	0.1822	0.1738	C
NB	0.2357	0.1721	0.3077	0.223	0.151	C
KNN	0.8299	0.5543	0.4966	0.4986	0.4958	A
LR	0.3504	0.2963	0.2818	0.3008	0.2792	B
RC	0.3322	0.2616	0.2634	0.2863	0.2549	B
SVM	0.3234	0.2767	0.2891	0.2983	0.2659	B
DT	0.7766	0.5671	0.5241	0.5353	0.5268	A
RF	0.8156	0.5799	0.5486	0.5298	0.5367	A
XGB	0.8495	0.6041	0.5703	0.5405	0.5521	A

Table 6.8: *Most common technical debt types approach performance*

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.6333	0.5142	0.0857	0.6333	0.151	B
NB	0.5894	0.4243	0.6816	0.5436	0.6042	B
KNN	0.9243	0.8045	0.7845	0.7772	0.7808	A
LR	0.7062	0.6574	0.6534	0.6801	0.6665	A
RC	0.6918	0.6272	0.6349	0.6635	0.6489	A
SVM	0.7134	0.6434	0.1425	0.4061	0.211	B
DT	0.9204	0.808	0.7926	0.7942	0.7934	A
RF	0.9355	0.8327	0.8196	0.808	0.8137	A
XGB	0.9394	0.8357	0.8311	0.7988	0.8146	A

Table 6.9: *Most common technical debt types approach tuned performance*

Table 6.9 presents the results obtained from the tuned performance analysis, as discussed in Section 6.1.4. The results are categorized into five clusters: Cluster A includes KNN, RC LR, DT, RF, and XGB; Cluster B includes DC, NB, and SVM.

Simple Approach

We employed a simple approach to determine when to pay off a technical debt item. In this approach, we applied machine learning methods directly to the priority labels and the complete dataset, resulting in one of the six categories: immediately, as soon as possible, in the next release, in the next few releases, when there is free time, or never.

Although more straightforward than the previous approaches, this simple approach provides a more realistic evaluation of the performance of ML methods in a context closer to real-world scenarios. It does not involve any pre-processing steps related to the number of classes or the quality of the collected data.

Table 6.10 shows the performance results of the simple approach using traditional analysis. We see that the best methods of classifying when TDI should be paid in general are KNN, DT, RF, and XGB (Cluster A), followed by NB, LR, RC, and SVM (Cluster B), and finally DC (Cluster C). We also see from the results that the performance of Cluster A methods is around four times better than the "random" method of Cluster C. In Cluster A,

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.1713	0.1527	0.1327	0.1619	0.1273	C
NB	0.2262	0.1653	0.3676	0.2054	0.1356	B
KNN	0.818	0.5527	0.4931	0.4914	0.4892	A
LR	0.3249	0.2943	0.3138	0.3142	0.283	B
RC	0.305	0.2534	0.2944	0.287	0.2474	B
SVM	0.3114	0.271	0.3194	0.3069	0.2577	B
DT	0.809	0.5508	0.5014	0.5035	0.5002	A
RF	0.814	0.5679	0.5145	0.5145	0.5127	A
XGB	0.8399	0.5943	0.5487	0.5203	0.5296	A

Table 6.10: *Simple approach performance*

accuracy ranged between 0.55 and 0.59, and F1 ranged between 0.49 and 0.53.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.6333	0.5169	0.8511	0.6333	0.1501	B
NB	0.5807	0.4235	0.7022	0.5448	0.6134	B
KNN	0.9202	0.8021	0.7766	0.7785	0.7775	A
LR	0.6824	0.6035	0.6615	0.6619	0.6617	B
RC	0.305	0.2534	0.2944	0.287	0.2474	B
SVM	0.6781	0.6038	0.6904	0.6754	0.6828	B
DT	0.9264	0.8088	0.7857	0.7903	0.788	A
RF	0.9306	0.8226	0.8076	0.7996	0.8036	A
XGB	0.9371	0.8251	0.8123	0.7922	0.8021	A

Table 6.11: *Simple approach tuned performance*

When we applied the tuned analysis to assess the performance of the machine learning methods, Table 6.11 displays that only two clusters were formed. Cluster A comprises KNN, DT, RF, and XGB, while Cluster B comprises DC, NB, RC, and SVM. Cluster A's accuracy ranged between 0.80 and 0.82, with F1 scores falling from 0.78 to 0.80.

The methods within the highest-performing cluster remained unchanged during the tuned analysis. However, the remaining three clusters from the traditional analysis merged into a single cluster in the tuned analysis. The methods within Cluster B exhibit an accuracy of approximately 0.55, and their F1 scores, precision, and recall follow a similar range. It is meaningful that Cluster A's accuracy is approximately 1.5 times better than Cluster B's.

2-layers Approach

Lastly, we developed a two-layers approach taking advantage of the binary machine learning classifier performance to decide whether to pay off or not a technical debt in the first level. Then, in the second layer, apply a method to decide when to make the payment only for cases where the first layer returns the decision for payment.

In the first layer, we apply the DT method with the values of the same hyperparameters used to infer either pay or not. If the prediction is 6 (Never), the method returns 6. Otherwise, it goes to the second layer. We chose DT because it is in the best-performing cluster (see Table 6.5), is more sophisticated than KNN, and takes much less training time than RF and XGB. In addition, we applied the SK analyses to cluster A of the Pay or Not label and the Simple Approach. The Pay or Not label is in Cluster A, and the Simple Approach is in Cluster B.

Moving on to the second layer, we applied one of the nine methods (including DT) to the data, excluding the instances predicted as Never in the first layer. This subset of data was utilized to predict the remaining five labels, indicating when to pay off the debt.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.1728	0.166	0.167	0.1667	0.1557	C
NB	0.2251	0.1615	0.37	0.2008	0.1289	B
KNN	0.8181	0.5519	0.4899	0.4905	0.4889	A
LR	0.3253	0.2912	0.3102	0.3108	0.2798	B
RC	0.3046	0.2527	0.2902	0.2836	0.2452	B
SVM	0.3163	0.3069	0.3407	0.3255	0.2952	B
DT	0.7768	0.5176	0.4691	0.4775	0.4704	A
RF	0.8268	0.5916	0.5403	0.5302	0.5336	A
XGB	0.8395	0.576	0.5256	0.5063	0.5131	A

Table 6.12: *2-layers approach performance*

Table 6.12 presents the performance results of the two-layers approach using traditional analysis. The findings reveal that the most effective methods for prioritizing when to pay off a technical debt item are KNN, RF, and XGB, which are classified under Cluster A. The performance of Cluster A methods surpasses that of the "random" DC method in Cluster C by approximately five times. Cluster A's accuracy ranged between 0.55 and 0.59, with F1 scores falling within the range of 0.49 to 0.53.

Method	Acc T/V	Accuracy	Precision	Recall	F1-score	Scott-Knott
DC	0.6353	0.4791	0.0798	0.6067	0.1411	C
NB	0.5833	0.4127	0.7148	0.5385	0.6138	B
KNN	0.9234	0.8033	0.7763	0.7809	0.7786	A
LR	0.6816	0.6037	0.6617	0.662	0.6618	B
RC	0.6613	0.5854	0.6505	0.6506	0.653	B
SVM	0.6781	0.6116	0.6778	0.6697	0.6737	A
DT	0.905	0.7963	0.7756	0.7857	0.7805	A
RF	0.9326	0.8252	0.813	0.7966	0.8047	A
XGB	0.9372	0.8247	0.8131	0.7912	0.802	A

Table 6.13: *2-layers approach tuned performance*

Table 6.13 provides the results obtained from the tuned performance analysis. The outcomes are categorized into three clusters. Cluster A (consisting of KNN, SVM, DT, RF, and XGB) demonstrated accuracy reaching 0.82, with F1 scores falling within 0.80.

Comparing the tuned analysis to the traditional analysis, we observe that the SVM method changed from Cluster B to Cluster A. Even though its accuracy is lower than other cluster methods, the precision, recall, and F1-score caused the method to change cluster. The mean of accuracies of algorithms within Cluster A was approximately 1.5 times better than the DC method in Cluster C, and their F1 scores mean were approximately five times better.

6.2.4 RQ2.3: Which are the best Machine Learning algorithms to prioritize technical debt?

Based on the results presented in the tables, we can conclude that KNN, DT, RF, and XGB are the best-performing methods for prioritizing the payment of technical debt items. Additionally, DT and RF consistently demonstrate superior performance across all approaches when determining when to make the payment.

Hence, DT and RF are the most effective machine learning algorithms for technical debt prioritization. Although their performance may vary depending on the chosen approach and the number of response classes and technical debt types, they consistently belong to the best-performing cluster and outperform random choices significantly.

6.3 Discussion of the Results

In this section, we discuss the findings presented in the previous section, which aimed to evaluate the effectiveness of well-known machine learning methods in determining whether and when a technical debt item should be paid off in real software projects.

We evaluated nine machine learning methods for the classification task of determining whether a TDI should be paid off. The results indicate that KNN, DT, RF, and XGB are suitable for this task and can assist in making payment decisions regarding technical debt items in real-world software projects.

Regarding predicting when a technical debt item should be paid off, we evaluated the nine machine learning methods using four approaches: the 3-classes approach, the most common TD types approach, the simple approach, and the 2-layers approach. Additionally, we performed two types of performance analysis: traditional and tuned.

The 3-classes approach, while reducing the granularity of the payment values, achieved high accuracy values of approximately 0.7 and precision, recall, and F1 scores of 0.65 in traditional analysis. On the other hand, in the tuned analysis, all methods are in the same cluster; even the XGB is almost 10% better for accuracy than the DC (random choice). The traditional analysis revealed that KNN, DT, RF, and XGB could effectively determine the priority of TDIs as high, medium, or low. However, for tuned analysis, all methods are similarly efficient, implying that any method, including a random choice, results in the same value.

The most common TD types approach focused on the dataset's best-represented technical debt types. The methods in the best-performing group attained an accuracy of around 0.58 and an F1-score of 0.53 in the traditional analysis and accuracy and an

F1-score of 0.76 in the tuned analysis. As in the 3-classes approach, KNN, DT, RF, and XGB emerged as the best-performing methods, with XGB demonstrating high performance in the traditional analysis.

The simple approach preserved the data on technical debt items of all types and maintained the full granularity of the response scale indicating when the item should be paid. The highest-performing machine learning methods in this approach achieved an average accuracy of 0.57 (0.81 in the tuned analysis), average precision of 0.51 (0.8 in the tuned analysis), and average recall of 0.51 (0.76 in the tuned analysis). Once again, KNN, DT, RF, and XGB stood out as the best performers, outperforming random choices by a factor of five.

The 2-layers approach takes advantage of the high performance of DT in classifying non-payment technical debt items (answer "Never") in the first (binary) approach. It then applies another machine learning method to differentiate among the remaining five classes. The best-performing cluster achieved an average accuracy of 0.56 and an F1-score of 0.5, which improved to 0.77 accuracy and 0.78 F1-score in the tuned analysis. Despite its greater complexity and additional computational effort, the 2-layers approach yielded results very close to the simple approach, suggesting that the added complexity may need to be more worthwhile.

Analyzing the machine learning method results allows us to conclude that at least KNN, DT, RF, and XGF can assist in prioritizing technical debt in determining whether it should be paid off and when with the highest accuracy, precision, recall, and F1 score. Moreover, cluster analysis demonstrates that high-performance methods consistently outperform a method that employs random choices.

6.4 Implications for Researchers and Practitioners

Our findings provide a foundation for future research to enhance machine learning methods and parameters, enabling even more precise determination of whether and when a technical debt item should be paid off. These technical debt prioritization methods developed in our study could also be applied in other domains that rely on prioritization, such as management research involving decision-making. Additionally, our approaches can be customized to specific contexts, enhanced through pre-processing data preparation steps, or integrated into existing technical debt management frameworks. These routes present promising opportunities for further exploration and development in this field.

For practitioners, our results can offer valuable support in the decision-making process regarding TDI payments. Improved prioritization can help avoid unnecessary payments, leading to enhanced development speed. Furthermore, optimizing the task of technical debt payment enables tackling the most critical items first, which have the most significant impact on code quality and future software development.

6.5 Threats to Validity

This study took careful measures to mitigate potential biases and misinterpretations. Below, we outline the steps we took to reduce threats to validity, discuss the remaining threats, and their potential impact on result interpretation.

Construct validity: To enhance construct validity, we conducted a pilot study with developers and improved the survey tool based on their feedback. However, the accuracy of responses could still be a concern. Practitioners looking to adopt our approaches should consider collecting their data to train models in their specific context. Using SonarQube as a tool for identifying technical debt items poses a potential threat, as it may not detect all types of technical debt. Other tools or additional data sources could be explored to address this limitation. We assumed equidistance between the labels to calculate the tuned metrics but we didn't validate this hypothesis.

External validity: Our dataset included a diverse set of projects, developers, and TDIs, which increases the generalizability of our findings. However, the generalization to non-public software projects and other programming languages may be limited. Replication studies in different settings, including commercial projects and other programming languages, would help enhance external validity.

Internal validity: The selection of independent variables could introduce a threat if they do not truly relate to the most appropriate payment time for technical debt items. We mitigated this using a comprehensive set of source code metrics widely used in the literature. However, omitting project and lifecycle metrics could present a potential threat. The non-uniform distribution of collected data was addressed through oversampling techniques to normalize the distribution while retaining data.

Reliability: We provide a research package[4] containing the dataset and a GitHub repository[5] containing the data preparation and ML model training and evaluation scripts to replicate the results with minimum effort.

To ensure reliability and facilitate replication, we provide a research package[6] containing the dataset and a GitHub repository[7] with data preparation, machine learning model training, and evaluation scripts. Researchers can replicate our results easily using these resources.

While we tried to mitigate threats to validity, we must consider these limitations when interpreting and applying our findings. Future research should explore alternative tools, consider additional data sources, replicate the study in different contexts, and validate the approaches using independent datasets.

[4] https://zenodo.org/record/7709535

[5] https://github.com/diogojpina/td-ml-prioritization

[6] https://zenodo.org/record/7709535

[7] https://github.com/diogojpina/td-ml-prioritization

6.6 Conclusion and Future Work

In this study, we developed technical debt prioritization methods using well-known machine learning algorithms to determine whether a technical debt item should be paid off and when the payment should be made. We collected data through a survey from 276 developers working on 207 Java public projects hosted on GitHub. Each survey respondent provided their opinion on when each item should be paid off, selecting from six response options.

Using the survey data and 27 source code metrics as features, we trained machine learning methods using a training-validation-test strategy. We evaluated various methods, including DC, NB, KNN, LR, RC, SVM, DT, RF, and XGB, to classify whether and when to pay off a technical debt item.

To determine whether to pay off a TDI, KNN, DT, RF, and XGB consistently achieved the highest performance, with an average accuracy of around 0.79. This performance was significantly better than a random choice, indicating that these methods effectively prioritize whether to pay off a TDI.

Regarding when to pay off a TDI, we applied four different approaches: 3-classes, common TD types, simple, and 2-layers. We evaluated the methods using both traditional and tuned analysis. Across all approaches, KNN, DT, RF, and XGB had the highest performance.

As we increase the number of labels or do not apply restrictions to the data, the performance of the methods decreases from an average accuracy of 0.7 to 0.57 in the traditional analysis and from 0.87 to 0.81 in the tuned analysis. These results show that using traditional analysis machine learning methods has good results compared to a random choice; however, it does not perform close to 90%, being efficient but ineffective. For the tuned analysis, KNN, DT, RF, and XGB methods proved to be efficient and effective.

The results indicate that the selected machine learning methods exceeded a random choice in prioritizing technical debt items across all approaches. However, future research should expand the dataset to include other programming languages and incorporate commercial software projects. Additionally, exploring alternative technical debt identification tools such as CAST and Squore could uncover new types of code technical debt. Investigating other types of technical debt, such as architectural and self-admitted debt, would also be valuable. Lastly, advanced artificial intelligence techniques, like deep learning, could be explored for prioritization.

By considering these directions for future research, we can further refine the technical debt prioritization methods and extend their applicability to a broader range of software projects and technical debt types.

Chapter 7

Conclusions

Although there are several works on technical debt prioritization in the literature, this is an open topic and needs more effort to make feasible prioritization in a more comprehensive and automated way. It would allow making-decision on which technical debt items to be paid off and in which order in an easy and integrated way into software development environments.

In this work, we perform two main studies that help to close the technical debt prioritization gaps. The first is a qualitative study to understand software developers' criteria for prioritizing technical debt items. The second is a quantitative study to develop machine learning methods to prioritize whether a technical debt item should be paid and when to pay.

In addition to the two major studies, we also performed a systematic literature mapping to find studies that directly or indirectly addressed the topic of technical debt prioritization. We also created a tool to mine software projects on GitHub and analyzed them in Sonar Qube to identify technical debt items.

7.1 Summary of Findings

In this section, we will revisit the research questions of this thesis to bring the main findings.

The first research question and its subquestions are:

- RQ1. How do developers prioritize technical debt?
  - RQ1.1. How do developers decide whether a code technical debt item should or should not be paid off?
  - RQ1.2 How do developers decide when a code technical debt item should be paid off?

The developers used a variety of criteria to prioritize whether a technical debt should be paid off and when. Depending on the situation, they had several motivations when deciding to pay off a technical debt item or not. We grouped these criteria into three super-categories to express when technical debt should not be paid off and two to express when an item should be paid off.

Every time a developer decided to pay off the technical debt, he wanted to make that soon: immediately, as soon as possible, or in the next release. However, it needs to be clarified from the data if this decision is technical or personal feeling based.

They also used distinct criteria for each priority level. When developers choose the same criterion in different cases, they use the same priority level or a neighbor one. There are only two code exceptions to that assertion, as there was more variety in developers' answers related to these criteria.

In addition, we found that each software project needs a specific set of rules to identify technical debt items that are relevant to the project. For most of the never answer, developers explained they were using their pattern resulting in a technical debt item in Sonar Qube.

Finally, we presented a framework to prioritize technical debt items based on the decision criteria of the respondents.

- RQ2. How to prioritize the payment of technical debt automatically?
  - RQ2.1. How effective are ML models for deciding **whether** or not a technical debt item should be paid?
  - RQ2.2. How effective are ML models for deciding **when** a technical debt item should be paid?
  - RQ2.3. Which are the best Machine Learning algorithms to prioritize technical debt?

We tested nine well-known machine learning methods over 27 features to build models to prioritize whether a technical debt item should be paid off. We found KNN, DT, RF, and XGB achieved the highest performance, averaging 0.86 for accuracy, 0.92 for precision, and 0.86 for recall. These results indicate that methods can be used to decide about pay or not a technical debt item.

We also tested the same nine machine learning methods over same 27 features to build models to prioritize when a technical debt item should be paid off. We applied four approaches: 3-classes, most common technical debt types, simple, and 2-layers, described in section 6.2.2. We assessed the performance of the methods using a traditional analysis with accuracy, precision, and recall around 0.85 for 3-classes and most common TD types and more than 0.80 for simple and 2-layers approach. We performed the tuned analysis and the metric results were around 0.95. KNN and RF achieved high-performance results in all four approaches. DT and XGB also achieved high performance for most of them.

Finally, KNN and RF are always in the cluster of methods that achieved the best performance for deciding whether and when a technical debt item should be paid. Also, DT and XGB are also presented in the best performance cluster for most of the approaches tested.

7.2 Future Work

For the technical debt prioritization criteria study, the main improvement is increasing the number of software projects and developers to cover more project kinds (industrial projects, distinct programming languages, and development paradigms, for instance) and types of technical debt (architectural and testing, for example).

Further research for the machine learning study should add other programming languages besides Java and include industrial software projects. They also could use other technical debt identification tools to find new types of technical debt not identified by Sonar Qube. Another possibility is adding other types of technical debt. Besides that, other artificial intelligence methods, such as deep learning, could be investigated to prioritize technical debt.

The Sonarlizer Xplorer tool could add new source code repositories integration, such as GitLab, BitBucket, and Source Forge. The mining tool also could collect other metrics and data related to the software project, such as commits and releases. In addition, Sonarlizer could add other code analysis tools to provide new code metrics and types of technical debt.

References

[ABAD and RUHE 2015] Zahra Shakeri Hossein ABAD and Guenther RUHE. "Using real options to manage technical debt in requirements engineering". In: *Requirements Engineering Conference (RE), 2015 IEEE 23rd International.* IEEE. 2015, pp. 230–235 (cit. on pp. 29, 31, 33, 34).

[AKBARINASAJI 2015] Shirin AKBARINASAJI. "Toward measuring defect debt and developing a recommender system for their prioritization". In: *Proceedings of the 13th International Doctoral Symposium on Empirical Software Engineering.* 2015, pp. 15–20 (cit. on pp. 29–32, 34, 35).

[ALBARAK and BAHSOON 2018] Mashel ALBARAK and Rami BAHSOON. "Prioritizing technical debt in database normalization using portfolio theory and data quality metrics". In: *Proceedings of the 2018 International Conference on Technical Debt.* 2018, pp. 31–40 (cit. on pp. 31, 32, 34).

[ALBUQUERQUE *et al.* 2022] Danyllo ALBUQUERQUE *et al.* "Comprehending the use of intelligent techniques to support technical debt management". In: *Proceedings of the International Conference on Technical Debt.* 2022, pp. 21–30 (cit. on p. 29).

[ALDAEEJ and SEAMAN 2018] Abdullah ALDAEEJ and Carolyn SEAMAN. "From lasagna to spaghetti: a decision model to manage defect debt". In: *2018 IEEE/ACM International Conference on Technical Debt (TechDebt).* IEEE. 2018, pp. 67–71 (cit. on pp. 31, 33, 34).

[ALFAYEZ and BOEHM 2019] Reem ALFAYEZ and Barry BOEHM. "Technical debt prioritization: a search-based approach". In: *2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS).* IEEE. 2019, pp. 434–445 (cit. on pp. 31, 32, 34).

[ALFAYEZ, WINN, *et al.* 2023] Reem ALFAYEZ, Robert WINN, Wesam ALWEHAIBI, Elaine VENSON, and Barry BOEHM. "How sonarqube-identified technical debt is prioritized: an exploratory case study". *Information and Software Technology* (2023), p. 107147 (cit. on pp. 29, 30).

[R. d. ALMEIDA 2019] Rodrigo de ALMEIDA. "Business-driven technical debt prioritization". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2019, pp. 605–609 (cit. on pp. 30, 31).

[R. d. Almeida *et al.* 2018] Rodrigo de Almeida, Uirá Kulesza, Christoph Treude, Aliandro Lima, *et al.* "Aligning technical debt prioritization with business objectives: a multiple-case study". In: *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2018, pp. 655–664 (cit. on pp. 29–31).

[R. R. d. Almeida *et al.* 2019] Rodrigo Rebouças de Almeida, Christoph Treude, and Uirá Kulesza. "Tracy: a business-driven technical debt prioritization framework". In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2019, pp. 181–185 (cit. on pp. 30, 31).

[Aniche 2012] M Aniche. *Repodriller.* 2012. url: https://github.com/mauricioaniche/repodriller (cit. on p. 45).

[Mauricio Aniche *et al.* 2020] Mauricio Aniche, Erick Maziero, Rafael Durelli, and Vinicius Durelli. "The effectiveness of supervised machine learning algorithms in predicting software refactoring". *IEEE Transactions on Software Engineering* (2020) (cit. on pp. 4, 71, 73).

[Araujo *et al.* 2011] Joao Eduardo M Araujo, Silvio Souza, and Marco Tulio Valente. "Study on the relevance of the warnings reported by java bug-finding tools". *IET software* 5.4 (2011), pp. 366–374 (cit. on p. 45).

[Ayewah *et al.* 2008] Nathaniel Ayewah, William Pugh, David Hovemeyer, J David Morgenthaler, and John Penix. "Using static analysis to find bugs". *IEEE software* 25.5 (2008), pp. 22–29 (cit. on p. 45).

[Barstow 1988] David Barstow. "Artificial intelligence and software engineering". In: *Exploring artificial intelligence.* Elsevier, 1988, pp. 641–670 (cit. on p. 2).

[Becker *et al.* 2018] Christoph Becker, Ruzanna Chitchyan, Stefanie Betz, and Curtis McCord. "Trade-off decisions across time in technical debt management: a systematic literature review". In: *Proceedings of the 2018 International Conference on Technical Debt.* ACM. 2018, pp. 85–94 (cit. on pp. 29, 30, 38).

[Bergstra and Bengio 2012] James Bergstra and Yoshua Bengio. "Random search for hyper-parameter optimization." *Journal of machine learning research* 13.2 (2012) (cit. on p. 73).

[Brauer *et al.* 2017] Johannes Brauer, Matthias Saft, Reinhold Plosch, and Christian Korner. "Improving object-oriented design quality: a portfolio-and measurement-based approach". In: *Proceedings of the 27th International Workshop on Software Measurement and 12th International Conference on Software Process and Product Measurement.* ACM. 2017, pp. 244–254 (cit. on pp. 29–32, 34, 35).

[Brown *et al.* 2010] Nanette Brown *et al.* "Managing technical debt in software-reliant systems". In: *Proceedings of the FSE/SDP workshop on Future of software engineering research.* ACM. 2010, pp. 47–52 (cit. on pp. 8, 9).

[Campbell and Papapetrou 2013] G Ann Campbell and Patroklos P Papapetrou. *SonarQube in action.* Manning Publications Co., 2013 (cit. on p. 45).

[Charalampidou *et al.* 2017] Sofia Charalampidou, Apostolos Ampatzoglou, Alexander Chatzigeorgiou, and Paris Avgeriou. "Assessing code smell interest probability: a case study". In: *Proceedings of the XP2017 Scientific Workshops.* ACM. 2017, p. 5 (cit. on pp. 22, 29, 31, 32, 34, 35).

[Chaturvedi and V. Singh 2012] Krishna Kumar Chaturvedi and VB Singh. "Determining bug severity using machine learning techniques". In: *2012 CSI sixth international conference on software engineering (CONSEG).* IEEE. 2012, pp. 1–6 (cit. on p. 71).

[Chatzigeorgiou *et al.* 2015] Alexander Chatzigeorgiou, Apostolos Ampatzoglou, Areti Ampatzoglou, and Theodoros Amanatidis. "Estimating the breaking point for technical debt". In: *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on.* IEEE. 2015, pp. 53–56 (cit. on pp. 29, 34).

[Choudhary and P. Singh 2016] Aabha Choudhary and Paramvir Singh. "Minimizing refactoring effort through prioritization of classes based on historical, architectural and code smell information". *CEUR Workshop Proceedings* 1771 (2016), pp. 76–79 (cit. on pp. 29, 31, 32, 34).

[Codabux and Williams 2016] Zadia Codabux and Byron J Williams. "Technical debt prioritization using predictive analytics". In: *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on.* IEEE. 2016, pp. 704–706 (cit. on pp. 29–32, 34).

[Costa *et al.* 2022] Alex Costa, Anna Marques, Ismayle Santos, and Rossana Andrade. "Towards a process to manage usability technical debts". In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering.* 2022, pp. 241–246 (cit. on pp. 29, 30).

[Cunnigham 1992] Ward Cunnigham. *Vídeo de Ward Cunnigham sobre dívida técnica, transcrito por June Kim e Lawrence Wang.* http://c2.com/cgi/wiki?WardExplainsDebtMetaphor. Acesso em 10 de Novembro de 2018. 1992 (cit. on p. 7).

[Curtis *et al.* 2012] Bill Curtis, Jay Sappidi, and Alexandra Szynkarski. "Estimating the size, cost, and types of technical debt". In: *Proceedings of the Third International Workshop on Managing Technical Debt.* IEEE Press. 2012, pp. 49–53 (cit. on pp. 10, 25).

[Da Silva *et al.* 2022] Filipe Da Silva, Ewertton De Souza, Rodrigo De Almeida, and Wylliams Santos. "Business-driven technical debt prioritization: a replication study". In: *2022 17th Iberian Conference on Information Systems and Technologies (CISTI).* IEEE. 2022, pp. 1–6 (cit. on p. 31).

[Daneva *et al.* 2013] Maya Daneva *et al.* “Agile requirements prioritization in large-scale outsourced system projects: an empirical study”. *Journal of systems and software* 86.5 (2013), pp. 1333–1353 (cit. on p. 29).

[De Almeida *et al.* 2021] Rodrigo De Almeida, Rafael Ribeiro, Christoph Treude, and Uirá Kulesza. “Business-driven technical debt prioritization: an industrial case study”. In: *2021 IEEE ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2021, pp. 74–83 (cit. on p. 31).

[De Toledo *et al.* 2022] Saulo Soares De Toledo, Antonio Martini, Phu Nguyen, and Dag Sjoberg. “Accumulation and prioritization of architectural debt in three companies migrating to microservices”. *IEEE Access* 10 (2022), pp. 37422–37445 (cit. on pp. 29, 30).

[Delamaro *et al.* 2007] Márcio Eduardo Delamaro, José Carlos Maldonado, and Mário Jino. *Introdução ao teste de software*. Editora Campus, 2007 (cit. on p. 13).

[Detofeno *et al.* 2022] Thober Detofeno, Andreia Malucelli, and Sheila Reinehr. “Priortd: a method for prioritization technical debt”. In: *Proceedings of the XXXVI Brazilian Symposium on Software Engineering*. 2022, pp. 230–240 (cit. on pp. 29, 31).

[El Naqa and Murphy 2015] Issam El Naqa and Martin J Murphy. “What is machine learning?” In: *machine learning in radiation oncology*. Springer, 2015, pp. 3–11 (cit. on p. 2).

[Ernst *et al.* 2015] Neil A Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L Nord, and Ian Gorton. “Measure it? manage it? ignore it? software practitioners and technical debt”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM. 2015, pp. 50–60 (cit. on pp. 8, 29, 30).

[Falessi, Shaw, *et al.* 2013] Davide Falessi, Michele A Shaw, Forrest Shull, Kathleen Mullen, and Mark Stein Keymind. “Practical considerations, challenges, and requirements of tool-support for managing technical debt”. In: *Managing Technical Debt (MTD), 2013 4th International Workshop on*. IEEE. 2013, pp. 16–19 (cit. on pp. 29, 30).

[Falessi and Voegele 2015] Davide Falessi and Alexander Voegele. “Validating and prioritizing quality rules for managing technical debt: an industrial case study”. In: *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE. 2015, pp. 41–48 (cit. on pp. 2, 29, 31, 32, 34, 35).

[Fernández-Sánchez, Díaz, *et al.* 2014] Carlos Fernández-Sánchez, Jessica Díaz, Jennifer Pérez, and Juan Garbajosa. “Guiding flexibility investment in agile architecting”. In: *System Sciences (HICSS), 2014 47th Hawaii International Conference on*. IEEE. 2014, pp. 4807–4816 (cit. on pp. 29, 31, 33, 34).

[Fernández-Sánchez, Garbajosa, *et al.* 2015] Carlos Fernández-Sánchez, Juan Garbajosa, and Agustin Yague. "A framework to aid in decision making for technical debt management". In: *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE. 2015, pp. 69–76 (cit. on pp. 29–32).

[Fontana *et al.* 2015] Francesca Arcelli Fontana, Vincenzo Ferme, Marco Zanoni, and Riccardo Roveda. "Towards a prioritization of code debt: a code smell intensity index". In: *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE. 2015, pp. 16–24 (cit. on pp. 29, 30, 33–35).

[Fowler 1999] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999 (cit. on p. 12).

[Fowler 2009] Martin Fowler. *Technical Debt Quadrant*. http://martinfowler.com/bliki/TechnicalDebtQuadrant.html. Acesso em 10 Novembro de 2018. 2009 (cit. on pp. viii, 9).

[A. Freire n.d.] Alexandre Freire. *Dívida Técnica: precisando de crédito? Ou "Como evitar que o cobrador bata na sua porta*. http://ccsl.ime.usp.br/pt-br/divida-tecnica-precisando-de-credito-ou-como-evitar-que-o-cobrador-bata-na-sua-porta. Acesso em 10 de Novembro de 2018 (cit. on p. 19).

[S. Freire, Rios, Gutierrez, *et al.* 2020] Sávio Freire, Nicolli Rios, Boris Gutierrez, *et al.* "Surveying software practitioners on technical debt payment practices and reasons for not paying off debt items". In: *Proceedings of the Evaluation and Assessment in Software Engineering*. 2020, pp. 210–219 (cit. on pp. 30, 31).

[S. Freire, Rios, Pérez, Castellanos, *et al.* 2023] Sávio Freire, Nicolli Rios, Boris Pérez, Camilo Castellanos, *et al.* "Software practitioners' point of view on technical debt payment". *Journal of Systems and Software* 196 (2023), p. 111554 (cit. on pp. 29, 30).

[S. Freire, Rios, Pérez, Torres, *et al.* 2021] Sávio Freire, Nicolli Rios, Boris Pérez, Darío Torres, *et al.* "How do technical debt payment practices relate to the effects of the presence of debt items in software projects?" In: *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2021, pp. 605–609 (cit. on pp. 29, 30).

[Garousi and Mäntylä 2016] Vahid Garousi and Mika V Mäntylä. "When and what to automate in software testing? a multi-vocal literature review". *Information and Software Technology* 76 (2016), pp. 92–117 (cit. on pp. 29, 30).

[Gomes *et al.* 2011] Rebeka Gomes *et al.* "An extraction method to collect data on defects and effort evolution in a constantly modified system". In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. 2011, pp. 27–30 (cit. on pp. 29, 30).

[Gousios 2013] Georgios Gousios. “The GHTorrent dataset and tool suite”. In: *Proceedings of the 10th Working Conference on Mining Software Repositories*. MSR. San Francisco, CA, May 2013, pp. 233–236. URL: http://www.gousios.gr/bibliography/G13.html (cit. on p. 45).

[Yuepu Guo and Seaman 2011] Yuepu Guo and Carolyn Seaman. “A portfolio approach to technical debt management”. In: *Proceedings of the 2nd Workshop on Managing Technical Debt*. ACM. 2011, pp. 31–34 (cit. on pp. 29, 31–34).

[Yuepu Guo, Seaman, *et al.* 2011] Yuepu Guo, Carolyn Seaman, *et al.* “Tracking technical debt—an exploratory case study”. In: *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*. IEEE. 2011, pp. 528–531 (cit. on p. 22).

[Yuepu Guo, Rodrigo Oliveira Spínola, *et al.* 2016] Yuepu Guo, Rodrigo Oliveira Spínola, and Carolyn Saman. “Exploring the costs of technical debt management–a case study”. *Empirical Software Engineering* 21.1 (2016), pp. 159–182 (cit. on pp. 1, 8, 10).

[Haendler *et al.* 2017] Thorsten Haendler, Stefan Sobernig, and Mark Strembeck. “Towards triaging code-smell candidates via runtime scenarios and method-call dependencies”. In: *Proceedings of the XP2017 Scientific Workshops*. ACM. 2017, p. 8 (cit. on pp. 29, 34).

[Harman 2012] Mark Harman. “The role of artificial intelligence in software engineering”. In: *2012 First International Workshop on Realizing AI Synergies in Software Engineering (RAISE)*. IEEE. 2012, pp. 1–6 (cit. on p. 2).

[Hormann *et al.* 2017] Peter Hormann *et al.* “Making ict decommissioning sexy!: challenges and opportunities”. *Australian Journal of Telecommunications and the Digital Economy* 5.2 (2017), p. 151 (cit. on pp. 29–31).

[Hunt and Hunt 2013] John Hunt and John Hunt. “Gang of four design patterns”. *Scala Design Patterns: Patterns for Practical Reuse and Design* (2013), pp. 135–136 (cit. on p. 13).

[Jayalakshmi and Santhakumaran 2011] T Jayalakshmi and A Santhakumaran. “Statistical normalization and back propagation for classification”. *International Journal of Computer Theory and Engineering* 3.1 (2011), pp. 1793–8201 (cit. on p. 73).

[Jelihovschi *et al.* 2014] Enio G Jelihovschi, José Cláudio Faria, and Ivan Bezerra Allaman. “Scottknott: a package for performing the scott-knott clustering algorithm in r”. *TEMA (São Carlos)* 15 (2014), pp. 3–17 (cit. on p. 76).

[Katin *et al.* 2022] Andrej Katin, Valentina Lenarduzzi, Davide Taibi, and Vladimir Mandic. “On the technical debt prioritization and cost estimation with sonarqube tool”. In: *Proceedings on 18th International Conference on Industrial Systems–IS’20: Industrial Innovation in Digital Age*. Springer. 2022, pp. 302–309 (cit. on p. 31).

[Kniberg 2013] Henrik Kniberg. *Good and Bad Technical Debt (and how TDD helps).* https://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt. Acesso em 10 de Novembro de 2018. 2013 (cit. on pp. viii, 20).

[Kouros *et al.* 2019] Panagiotis Kouros *et al.* "Jcaliper: search-based technical debt management". In: *Proceedings of the 34th ACM/SIGAPP Symposium on applied computing.* 2019, pp. 1721–1730 (cit. on pp. 31, 32, 34).

[Kruchten *et al.* 2012] Philippe Kruchten, Robert L Nord, and Ipek Ozkaya. "Technical debt: from metaphor to theory and practice". *Ieee software* 29.6 (2012), pp. 18–21 (cit. on p. 2).

[Leppanen *et al.* 2015] Marko Leppanen *et al.* "Decision-making framework for refactoring". In: *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on.* IEEE. 2015, pp. 61–68 (cit. on pp. 29–32, 38, 60).

[Letouzey 2012a] Jean-Louis Letouzey. *The SQALE Method - Definition Document.* 2012 (cit. on pp. viii, 15–20).

[Letouzey 2012b] Jean-Louis Letouzey. "The sqale method for evaluating technical debt". In: *Proceedings of the Third International Workshop on Managing Technical Debt.* IEEE Press. 2012, pp. 31–36 (cit. on p. 14).

[B. d. Lima *et al.* 2022] Bruno de Lima, Rogerio Garcia, and Danilo Eler. "Toward prioritization of self-admitted technical debt: an approach to support decision to payment". *Software Quality Journal* 30.3 (2022), pp. 729–755 (cit. on p. 31).

[*Managing Technical Debt* 2016] *Managing Technical Debt.* https://mtd2016dagstuhl.org/. Acesso em 10 Novembro de 2018. Dagstuhl Seminar, 2016 (cit. on p. 8).

[Mandic *et al.* 2021] Vladimir Mandic *et al.* "Technical and nontechnical prioritization schema for technical debt: voice of td-experienced practitioners". *IEEE Software* 38.6 (2021), pp. 50–58 (cit. on pp. 29, 31).

[Mann and Whitney 1947] Henry B Mann and Donald R Whitney. "On a test of whether one of two random variables is stochastically larger than the other". *The annals of mathematical statistics* (1947), pp. 50–60 (cit. on p. 69).

[Martini and Bosch 2015a] Antonio Martini and Jan Bosch. "The danger of architectural technical debt: contagious debt and vicious circles". In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on.* IEEE. 2015, pp. 1–10 (cit. on pp. 29, 30).

[Martini and Bosch 2015b] Antonio Martini and Jan Bosch. "Towards prioritizing architecture technical debt: information needs of architects and product owners". In: *Software Engineering and Advanced Applications (SEAA), 2015 41st Euromicro Conference on.* IEEE. 2015, pp. 422–429 (cit. on pp. 29, 30).

[Martini and Bosch 2016] Antonio Martini and Jan Bosch. "An empirically developed method to aid decisions on architectural technical debt refactoring: anacondebt". In: *Software Engineering Companion (ICSE-C), IEEE/ACM International Conference on*. IEEE. 2016, pp. 31–40 (cit. on pp. 29–32).

[Martini and Bosch 2017] Antonio Martini and Jan Bosch. "On the interest of architectural technical debt: uncovering the contagious debt phenomenon". *Journal of Software: Evolution and Process* 29.10 (2017) (cit. on pp. 29–32).

[Martini, Bosch, and Chaudron 2015] Antonio Martini, Jan Bosch, and Michel Chaudron. "Investigating architectural technical debt accumulation and refactoring over time: a multiple-case study". *Information and Software Technology* 67 (2015), pp. 237–253 (cit. on pp. 29–32).

[McConnell 2007] Steve McConnell. *Managing Technical Debt*. http://www.construx.com/10x_Software_Development/Technical_Debt. Acesso em 10 Novembro de 2018. 2007 (cit. on p. 8).

[Mensah *et al.* 2018] Solomon Mensah, Jacky Keung, Jeffery Svajlenko, Kwabena Ebo Bennin, and Qing Mi. "On the value of a prioritization scheme for resolving self-admitted technical debt". *Journal of Systems and Software* 135 (2018), pp. 37–54 (cit. on pp. 29, 31, 34).

[*Microsoft GHCrawler* n.d.] *Microsoft GHCrawler*. url: https://github.com/Microsoft/ghcrawler (cit. on p. 45).

[Mitchell *et al.* 1997] Tom M Mitchell *et al.* "Machine learning" (1997) (cit. on p. 2).

[Mohammed *et al.* 2020] Roweida Mohammed, Jumanah Rawashdeh, and Malak Abdullah. "Machine learning with oversampling and undersampling techniques: overview study and experimental results". In: *2020 11th international conference on information and communication systems (ICICS)*. IEEE. 2020, pp. 243–248 (cit. on p. 71).

[Mohan *et al.* 2016] Michael Mohan, Des Greer, and Paul McMullan. "Technical debt reduction using search based automated refactoring". *Journal of Systems and Software* 120 (2016), pp. 183–194 (cit. on pp. 29–32, 34, 35).

[Morgenthaler *et al.* 2012] J David Morgenthaler, Misha Gridnev, Raluca Sauciuc, and Sanjay Bhansali. "Searching for build debt: experiences managing technical debt at google". In: *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press. 2012, pp. 1–6 (cit. on pp. 29, 30).

[Pérez *et al.* 2021] Boris Pérez *et al.* "Technical debt payment and prevention through the lenses of software architects". *Information and Software Technology* 140 (2021), p. 106692 (cit. on p. 29).

[Petersen *et al.* 2008] Kai Petersen, Robert Feldt, Shahid Mujtaba, and Michael Mattsson. “Systematic mapping studies in software engineering.” In: *EASE*. Vol. 8. 2008, pp. 68–77 (cit. on pp. viii, 25, 26, 28, 37).

[Pina and Goldman 2016] Diogo Pina and Alfredo Goldman. “Gerenciando dívida técnica: estado atual e novas propostas em métodos de medida”. In: *Dissertação (Mestrado)*. USP. 2016 (cit. on p. 1).

[Pina, Goldman, and Seaman 2022] Diogo Pina, Alfredo Goldman, and Carolyn Seaman. “Sonarlizer xplorer: a tool to mine github projects and identify technical debt items using sonarqube”. In: *2022 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2022, pp. 71–75 (cit. on pp. 63, 64).

[Pina, Goldman, and Tonin 2021] Diogo Pina, Alfredo Goldman, and Graziela Tonin. “Technical debt prioritization: taxonomy, methods results, and practical characteristics”. In: *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2021, pp. 206–213 (cit. on p. 25).

[Pina, Seaman, *et al.* 2022] Diogo Pina, Carolyn Seaman, and Alfredo Goldman. “Technical debt prioritization: a developer’s perspective”. In: *2022 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2022, pp. 46–55 (cit. on pp. 29, 30).

[Plösch *et al.* 2018] Reinhold Plösch, Johannes Brauer, Matthias Saft, and Christian Korner. “Design debt prioritization: a design best practice-based approach”. In: *2018 IEEE/ACM International Conference on Technical Debt (TechDebt)*. IEEE. 2018, pp. 95–104 (cit. on pp. 29–32, 34).

[Razali, Wah, *et al.* 2011] Nornadiah Mohd Razali, Yap Bee Wah, *et al.* “Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests”. *Journal of statistical modeling and analytics* 2.1 (2011), pp. 21–33 (cit. on p. 69).

[Refaeilzadeh *et al.* 2009] Payam Refaeilzadeh, Lei Tang, and Huan Liu. “Cross-validation.” *Encyclopedia of database systems* 5 (2009), pp. 532–538 (cit. on p. 73).

[L. Ribeiro *et al.* 2016] L Ribeiro, M Farias, M Mendonça, and R Spínola. “Decision criteria for the payment of technical debt in software projects: a systematic mapping study.” In: *ICEIS (1)*. 2016, pp. 572–579 (cit. on p. 60).

[L. F. Ribeiro *et al.* 2017] Leilane Ferreira Ribeiro, Nicolli Souza Rios Alves, Manoel Gomes de Mendonca Neto, and Rodrigo Oliveira Spínola. “A strategy based on multiple decision criteria to support technical debt management”. In: *Software Engineering and Advanced Applications (SEAA), 2017 43rd Euromicro Conference on*. IEEE. 2017, pp. 334–341 (cit. on pp. 29, 31, 34).

[RIEGEL and DOERR 2015] Norman RIEGEL and Joerg DOERR. “A systematic literature review of requirements prioritization criteria”. In: *International Working Conference on Requirements Engineering: Foundation for Software Quality*. Springer. 2015, pp. 300–317 (cit. on pp. 29, 30, 60).

[SCHMID 2013] Klaus SCHMID. “A formal approach to technical debt decision making”. In: *Proceedings of the 9th international ACM Sigsoft conference on Quality of software architectures*. ACM. 2013, pp. 153–162 (cit. on pp. 29, 30, 34).

[SEAMAN and Yuepo GUO 2011] Carolyn SEAMAN and Yuepo GUO. “Measuring and monitoring technical debt”. *Advances in Computers* 82.6810 (2011), pp. 25–46 (cit. on pp. viii, 1, 7, 8, 10, 11, 15).

[SEAMAN, Yuepu GUO, *et al.* 2012] Carolyn SEAMAN, Yuepu GUO, *et al.* “Using technical debt data in decision making: potential decision approaches”. In: *Proceedings of the Third International Workshop on Managing Technical Debt*. IEEE Press. 2012, pp. 45–48 (cit. on pp. 22, 29, 30).

[SHAPIRO and WILK 1965] Samuel Sanford SHAPIRO and Martin B WILK. “An analysis of variance test for normality (complete samples)”. *Biometrika* 52.3/4 (1965), pp. 591–611 (cit. on p. 69).

[SKOURLETOPOULOS, CHATZIMISIOS, *et al.* 2015] Georgios SKOURLETOPOULOS, Periklis CHATZIMISIOS, *et al.* “A fluctuation-based modelling approach to quantification of the technical debt on mobile cloud-based service level”. In: *Globecom Workshops (GC Wkshps), 2015 IEEE*. IEEE. 2015, pp. 1–6 (cit. on pp. 29, 30, 34, 35).

[SKOURLETOPOULOS, MAVROMOUSTAKIS, *et al.* 2016] Georgios SKOURLETOPOULOS, Constandinos MAVROMOUSTAKIS, *et al.* “Quantifying and evaluating the technical debt on mobile cloud-based service level”. In: *Communications (ICC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 1–7 (cit. on pp. 29, 31, 34).

[SNIPES *et al.* 2012] Will SNIPES, Brian ROBINSON, Yuepu GUO, and Carolyn SEAMAN. “Defining the decision factors for managing defects: a technical debt perspective”. In: *Managing Technical Debt (MTD), 2012 Third International Workshop on*. IEEE. 2012, pp. 54–60 (cit. on pp. 29, 31, 34).

[M. STOCHEL *et al.* 2022] Marek STOCHEL, Piotr CHOLDA, and Mariusz WAWROWSKI. “Adopting devops paradigm in technical debt prioritization and mitigation”. In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2022, pp. 306–313 (cit. on pp. 29, 30).

[M. G. STOCHEL, CHOLDA, *et al.* 2020] Marek G STOCHEL, Piotr CHOLDA, and Mariusz R WAWROWSKI. “Continuous debt valuation approach (codva) for technical debt prioritization”. In: *2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE. 2020, pp. 362–366 (cit. on pp. 30, 31).

[M. G. Stochel, Wawrowski, *et al.* 2022] Marek G Stochel, Mariusz Wawrowski, and Piotr Cholda. “Technical debt prioritization in telecommunication applications: why the actual refactoring deviates from the plan and how to remediate it? case study in the covid era”. *Applied Sciences* 12.22 (2022), p. 11347 (cit. on p. 31).

[Stol *et al.* 2016] K Stol, P Ralph, and B Fitzgerald. “Grounded theory in software engineering research: a critical review and guidelines”. In: *ICSE*. 2016, pp. 120–131 (cit. on p. 52).

[Strauss and Corbin 1994] A Strauss and J Corbin. “Grounded theory methodology: an overview.” (1994) (cit. on p. 52).

[Anselm Strauss and Juliet Corbin 1998] Anselm Strauss and Juliet Corbin. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage 2nd Ed, 1998 (cit. on pp. 52, 57).

[Tornhill 2018] Adam Tornhill. “Prioritize technical debt in large-scale systems using codescene”. In: *Proceedings of the 2018 International Conference on Technical Debt.* 2018, pp. 59–60 (cit. on pp. 31, 32, 34).

[Tsintzira *et al.* 2020] Angeliki-Agathi Tsintzira, Elvira-Maria Arvanitou, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. “Applying machine learning in technical debt management: future opportunities and challenges”. In: *Quality of Information and Communications Technology: 13th International Conference, QUATIC 2020, Faro, Portugal, September 9–11, 2020, Proceedings 13.* Springer. 2020, pp. 53–67 (cit. on pp. 29, 30).

[Tsoukalas, Mittas, *et al.* 2021] Dimitrios Tsoukalas, Nikolaos Mittas, *et al.* “Machine learning for technical debt identification”. *IEEE Transactions on Software Engineering* (2021) (cit. on pp. 4, 38, 71, 73).

[Tsoukalas, Siavvas, *et al.* 2023] Dimitrios Tsoukalas, Miltiadis Siavvas, Dionysios Kehagias, Apostolos Ampatzoglou, and Alexander Chatzigeorgiou. “A practical approach for technical debt prioritization based on class-level forecasting”. *Journal of Software: Evolution and Process* (2023), e2564 (cit. on p. 31).

[Wiese *et al.* 2022] Marion Wiese, Paula Rachow, Matthias Riebisch, and Julian Schwarze. “Preventing technical debt with the tap framework for technical debt aware management”. *Information and Software Technology* 148 (2022), p. 106926 (cit. on pp. 29–31).

[Yang and Shami 2020] Li Yang and Abdallah Shami. “On hyperparameter optimization of machine learning algorithms: theory and practice”. *Neurocomputing* 415 (2020), pp. 295–316 (cit. on pp. 71, 73).

[Yli-Huumo *et al.* 2016] Jesse Yli-Huumo, Andrey Maglyas, Kari Smolander, Johan Haller, and Hannu Törnroos. "Developing processes to increase technical debt visibility and manageability–an action research study in industry". In: *International Conference on Product-Focused Software Process Improvement.* Springer. 2016, pp. 368–378 (cit. on pp. 29–31, 34, 35).

[Zazworka, Izurieta, *et al.* 2014] Nico Zazworka, Clemente Izurieta, *et al.* "Comparing four approaches for technical debt identification". *Software Quality Journal* 22.3 (2014), pp. 403–426 (cit. on pp. 10, 12).

[Zazworka and Seaman 2013] Nico Zazworka and Carolyn Seaman. *Identifying and Managing Technical Debt.* http://www.slideshare.net/zazworka/identifying-and-managing-technical-debt. Acesso em 10 de Novembro de 2018. 2013 (cit. on pp. 11, 13).

[Zazworka, Seaman, and Shull 2011] Nico Zazworka, Carolyn Seaman, and Forrest Shull. "Prioritizing design debt investment opportunities". In: *Proceedings of the 2nd Workshop on Managing Technical Debt.* ACM. 2011, pp. 39–42 (cit. on pp. 29–31, 33, 34).

[Zazworka, Rodrigo O Spínola, *et al.* 2013] Nico Zazworka, Rodrigo O Spínola, Antonio Vetro, Forrest Shull, and Carolyn Seaman. "A case study on effectively identifying technical debt". In: *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering.* ACM. 2013, pp. 42–47 (cit. on p. 10).