UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

José Laerte Pires Xavier Júnior

Documenting and Managing Self-Admitted Technical Debt Using Issues

Belo Horizonte
2022

José Laerte Pires Xavier Júnior

Documenting and Managing Self-Admitted Technical Debt Using Issues

Final Version

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio Valente

Belo Horizonte
2022

Xavier Júnior, José Laerte Pires.

X3d Documenting and managing self-admitted technical debt using issues [recurso eletrônico] :/ José Laerte Pires Xavier Júnior –2022.
1 recurso online (112 f. il, color.) : pdf.

Orientador: Marco Túlio Oliveira Valente.

Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciências da Computação.
Referências: f.104-112

1. Computação – Teses. 2. Dívida técnica – Teses. 3. Mineração de repositórios de software – Teses. 4. Framework (Programa de computador) – Teses. 5. Software - Documentação – Teses. I. Valente, Marco Túlio Oliveira. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Computação. III.Título.

CDU 519.6*32(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Documenting and Managing Self-Admitted Technical Debt Using Issues

JOSÉ LAERTE PIRES XAVIER JÚNIOR

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

PROF. MARCO TÚLIO DE OLIVEIRA VALENTE - Orientador
Departamento de Ciência da Computação - UFMG

[illegible]

PROF. ALFREDO GOLDMAN VEL LEJBMAN
Departamento de Ciência da Computação - USP

[illegible]

PROF. RODRIGO OLIVEIRA SPÍNOLA
College of Engineering - Virginia Commonwealth University

André Cavalcante Hora

PROF. ANDRÉ CAVALCANTE HORA
Departamento de Ciência da Computação - UFMG

Eduardo Figueiredo

PROF. EDUARDO MAGNO LAGES FIGUEIREDO
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 15 de Dezembro de 2022.

To Lucy and Laerte, my dearly loved parents.

Acknowledgments

Esta tese é fruto de uma intensa jornada de aprendizado. Por isso, agradeço a cada um dos familiares, amigos e colegas que tanto me ensinaram e incentivaram ao longo desse caminho. Agradeço, em especial:

À Deus, que me molda e me capacita para ter um coração aprendiz.

Aos meus pais, meu irmão, minha cunhada e meu sobrinho, que me ensinam diariamente o valor da família e o significado das palavras amor e saudade.

Aos meus avós e demais familiares, que sempre me inspiraram a ir além.

À Derek, Jordana e demais amigos de longa data, que recordam constantemente as minhas raízes e não permitem que eu me perca de mim mesmo, por me ensinarem que as sextas-feiras sempre chegarão.

Aos meus novos-velhos amigos de BH, que durante esses anos formaram a minha segunda família, por me ensinarem que casa é o lugar onde o coração habita.

Aos colegas da PUC Minas e do ASERG, pelo aprendizado constante e pelos momentos de crescimento pessoal e profissional. À Fábio Ferreira, Aline Brito, Rodrigo Brito e João Eduardo Montandon, pelas parcerias de trabalho.

Aos meus alunos e ex-alunos, por serem meu combustível durante a construção deste trabalho. Por tudo que aprendi nesses anos de troca em sala de aula.

Aos meus ex-professores, pela formação que moldou a minha mente, o meu comportamento e a minha personalidade. Pelas memórias das boas aulas e cuidadosas orientações, que constantemente me servem de inspiração.

Ao meu orientador Marco Túlio, pela acolhida, atenção e paciência em conduzir este trabalho. Pelas revisões, sugestões e suporte essenciais durante todos esses anos. Por me ensinar tanto, de tantas maneiras.

Aos membros da banca, Alfredo Goldman, Rodrigo Spínola, André Hora e Eduardo Figueiredo, pela disponibilidade em contribuir com este trabalho.

À Prodemge e aos times ProBPMS e SIGAF JUD, pela parceria e suporte, essenciais para a conclusão do trabalho.

Ao DCC/UFMG e a CAPES, pelo suporte financeiro, logístico e profissional.

“*It is our choices, Harry, that show what we truly are, far more than our abilities.*”

(J.K. Rowling)

Resumo

A metáfora da Dívida Técnica (TD, do inglês *Technical Debt*) refere-se aos custos inequívocos de manutenção e evolução gerados por decisões sub-ótimas comumente tomadas por desenvolvedores de software. Desde a sua concepção, o termo foi rapidamente adotado na indústria e tornou-se objeto de diversos estudos, investigando técnicas de identificação, gerenciamento e pagamento de TD. Nos últimos anos, vários estudos surgiram em torno do fato de que desenvolvedores documentam explicitamente suas dívidas, o que a literatura conhece como Dívida Técnica Auto-Admitida (SATD, do inglês *Self-Admitted Technical Debt*). Particularmente, a maioria dos estudos anteriores investigou essa prática analisando comentários de código-fonte para identificar indícios de admissão de TD. Nesta tese, denotamos essa forma de SATD como SATD-C. No entanto, ainda não está claro como diferentes artefatos de software são utilizados para admitir TD. Especificamente, poucos estudos investigaram profundamente SATD em sistemas de rastreamento de *issues*, apesar da sua crescente adoção no desenvolvimento de software. Para contribuir na resolução desse problema, descreve-se nesta tese um estudo aprofundado sobre a adoção de *issues* para documentar e gerir a Dívida Técnica Auto-Admitida (denotado como SATD-I). Esta pesquisa está organizada em quatro unidades de trabalho. Inicialmente, são exploradas as características da adoção do SATD-I, em termos dos tipos de TD comumente documentados em *issues*, e as motivações para a criação e o pagamento delas. Em seguida, estende-se os resultados iniciais investigando a interação entre SATD-C e SATD-I em um *dataset* de larga escala. Com base nas evidências de que elas possuem naturezas distintas, avaliam-se as circunstâncias em que cada forma de SATD é mais adequada. Finalmente, reune-se o conhecimento produzido nos estudos empíricos anteriores para propor e avaliar um *framework* leve para apoiar o gerenciamento de TD através da criação de *issues*. Esse *framework* é denominado LTD: *Less Technical Debt Framework*. No geral, os resultados obtidos confirmam que os desenvolvedores também utilizam *issues* para admitir TD em seus projetos. Mostra-se também que *issues* são mais adequadas para documentar dívidas de alto nível e alta prioridade. Por fim, foram obtidos resultados promissores após a adoção do LTD em duas equipes de desenvolvimento em uma grande empresa pública. Por exemplo, as equipes conseguiram reduzir o TD e criar uma lista de *issues* para gerenciar o TD durante a execução dos sprints.

Palavras-chave: dívida técnica, documentação, mineração de repositórios de software, sistemas de rastreamento de issues, framework LTD.

Abstract

The Technical Debt (TD) metaphor refers to the unavoidable maintenance and evolution costs of the *not-quite-right* decisions commonly taken by software developers. The term was quickly adopted in industry and became the subject of various studies in the literature, mostly regarding techniques for TD identification, management and payment. In recent years, several studies emerged around the observation that developers explicitly document their debts, which the literature refers as Self-Admitted Technical Debt (SATD). Particularly, most previous studies investigated this practice by analyzing source code comments as indications of TD admission. In this thesis, we denote this form of SATD as SATD-C. However, there is a lack of knowledge about the adoption of different artifacts to admit TD. Particularly, few studies deeply investigated SATD in issue tracker systems, despite the increasing adoption of issues in software development. In order to contribute to this problem, we describe in this thesis an in-depth study on the adoption of issues to document and manage Self-Admitted Technical Debt (which we refer as SATD-I). We organize the research in four major working units. We start by exploring the characteristics of SATD-I adoption, in terms of the types of TD commonly documented in issues, and the motivations of its insertion and payment. Next, we strengthen our initial results by investigating the interplay between SATD-C and SATD-I in a large-scale dataset. Based on the evidences that they have distinct natures, we assess the circumstances when each form is more suitable. Finally, we wrap-up the knowledge produced in our empirical studies by proposing and evaluating a light-weight framework to support developers to manage TD through the creation of issues. We call this framework as LTD: Less Technical Debt Framework. Overall, our results confirm that developers also use issues to admit TD in their projects. We also show that issues are more suitable to document high-level and high-priority debts, mostly regarding concerns that need visibility and discussions between developers. Finally, we achieved promising results after adopting LTD in two development teams in a large public company. For example, the teams could reduce TD and create a backlog of issues to manage TD during the execution of sprints.

Keywords: technical debt, documentation, mining software repositories, issue tracking systems, LTD framework.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

We start this chapter by introducing the problem that motivates this thesis (Section 1.1). Then, we highlight the major goals and contributions obtained around our work (Section 1.2). Finally, we present the list of publications resulted from this thesis (Section 1.3), as well as the outline of the remaining of this document (Section 1.4).

1.1 Problem and Motivation

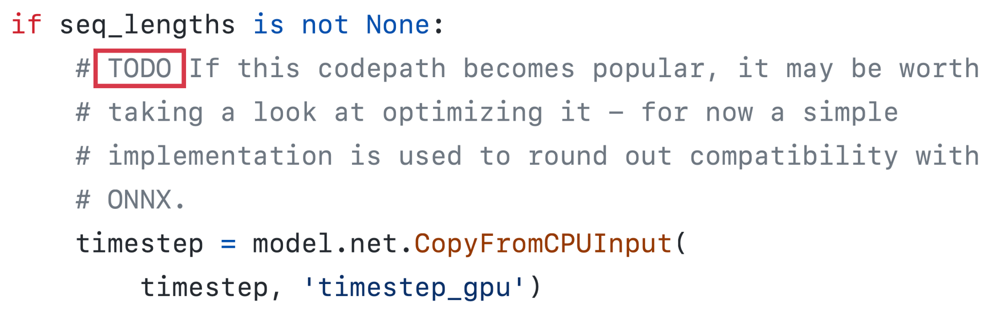
Modern software developers are under constant pressure to evolve their systems, in order to preserve existing clients or to explore new markets. During this process, it is inevitable to incur in sub-optimal technical decisions, whose accumulation will eventually emerge in the form of features that are more risky and difficult to implement. To frame this practice, Cunningham [1992] coined the **Technical Debt (TD)** metaphor:

> *Although immature code may work fine and be completely acceptable to the customer, excess quantities will make a program unmasterable, leading to extreme specialization of programmers and finally an inflexible product. Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite. Objects make the cost of this transaction tolerable. The danger occurs when the debt is not repaid. Every minute spent on not-quite-right code counts as interest on that debt.* (Cunningham [1992])

In fact, the term has been widely adopted since its definition [Kruchten et al., 2012] and became subject of various studies, mostly regarding its identification [Liu et al., 2018; Rios et al., 2018; Alves et al., 2016], management [Lim et al., 2012; Storey et al., 2008; Sierra et al., 2019a] and assessment [Wehaibi et al., 2016; Kamei et al., 2016; Silva et al., 2016]. In this context, the majority of these studies were based on the occurrence of code smells as a proxy to indicate TD. Moreover, they mostly focused on exploring business aspects that characterize the circumstances which companies introduce TD and prioritize the repayment of particular types of debts.

For example, Zazworka et al. [2011] first focused on the detection of Technical Debt, comparing the efficiency of automated tools against human assessments. In a follow-up study, Zazworka et al. [2013] measured the impact of Technical Debt in software quality, showing that God Classes have a higher impact on the overall measures. Fontana et al. [2012] also investigated the occurrence of TD in the form of code smells, proposing an approach to classify the debts based on their potential risk. Ernst et al. [2015] conducted a survey with 1.8K developers and found that architectural decisions are the most important cause of TD.

Due to its importance to the software process and quality, it is not a surprise to observe that developers also create explicit documentation to manage TD. Based on this practice, Potdar and Shihab [2014] took a novel direction from TD studies and coined the concept of **Self-Admitted Technical Debt (SATD)**. In this context, they used code comments as proxy to identity TD admission. For example, they showed that developers use terms such as *TODO*, *fixme*, and *hack* in comments to remind themselves or other developers that a given part of the code should be changed and improved in future sprints. As an example, Figure 1.1 illustrates an instance of SATD extracted from PYTORCH/PYTORCH source code.



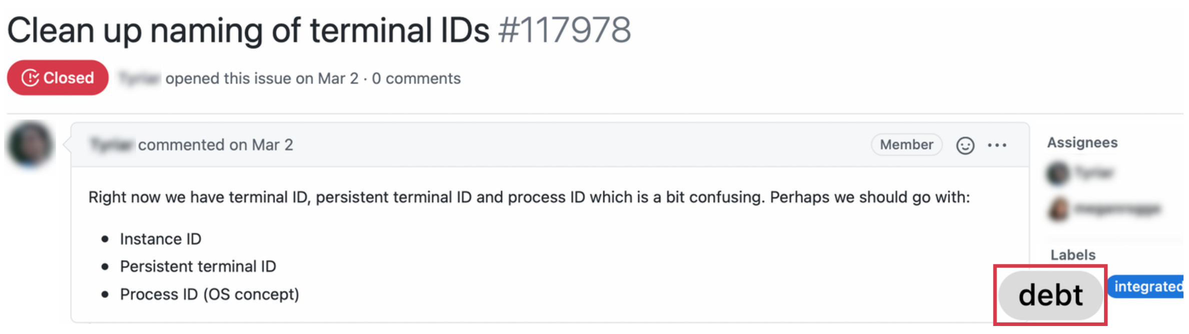
```
if seq_lengths is not None:
    # TODO If this codepath becomes popular, it may be worth
    # taking a look at optimizing it - for now a simple
    # implementation is used to round out compatibility with
    # ONNX.
    timestep = model.net.CopyFromCPUInput(
        timestep, 'timestep_gpu')
```

Figure 1.1: Example of SATD in PYTORCH/PYTORCH (see the *TODO* term in the top left corner).

Most previous studies on SATD followed this path, using code comments to identify TD and deepening the understanding about this form of debt. For example, Bavota and Russo [2016] used a larger dataset to replicate the original study conducted by Potdar and Shihab [2014]. They confirmed the usage of code comments as means to admit TD and observed a similar behavior in comparison to previously studied TD instances. Maldonado et al. [2017a] showed that most SATD instances are self-removed—paid by the same developer who introduced them—as part of bug fixing and enhancement activities. Zampetti et al. [2018] investigated the aforementioned removal operations to qualitatively characterize how SATD is paid. Sierra et al. [2019b] showed that SATD can be used as an indicator of architectural divergences.

In contrast to a plethora of studies performed on SATD, few studies relied on other artifacts to identify Technical Debt admission. Particularly, they explored the existence of TD admission in issue trackers by analyzing developers discussions and other textual interactions. For example, Bellomo et al. [2016] prospected this idea by manually examining a sample of 1,264 issues mined from four industry and governmental issue trackers. They found that developers discussed TD in 109 issues. Dai and Kruchten [2017] also studied this possibility by applying natural language processing and machine learning techniques to identify TD in 8,149 issues. As a result, the authors provided a set of 114 keywords that can be used to detect different types of TD from issue descriptions. Lastly, Li et al. [2020] conducted a case study to investigate the existence of SATD in issue discussions. They manually investigated a sample of 500 issues and identified a set of 117 discussion excerpts about TD.

In this thesis, **we follow this less-studied direction by making an in-depth investigation on the adoption of issues to identify, admit, and document TD**. In this context, we refine the state-of-the-art by arguing that TD in issues can also be indicated by the adoption of labels such as *technical debt*, *debt*, and *workaround*. As an example, Figure 1.2 illustrates this practice. Particularly, the issue in this figure highlights the need of cleaning part of the code of MICROSOFT/VSCODE to better handle similar terminal ID names. As we can see, it received a *debt* label.


Clean up naming of terminal IDs #117978

Closed opened this issue on Mar 2 · 0 comments

commented on Mar 2 Member

Right now we have terminal ID, persistent terminal ID and process ID which is a bit confusing. Perhaps we should go with:

- Instance ID
- Persistent terminal ID
- Process ID (OS concept)

Figure 1.2: Example of TD in MICROSOFT/VSCODE documented using an issue (see the *debt* label in the bottom right corner).

In this thesis, we assume that there are two types of SATD[1]:

1. **SATD documented using source code comments (SATD-C)**, which has been extensively studied in the past;

2. **SATD documented using issues (SATD-I)**, which has been less studied in the literature, usually only in a restrict set of textual interactions (issue descriptions and discussions).

[1]In fact, SATD can also be documented in other artifacts, such as commit messages, wikis, forum discussions, etc. However, these other artifacts are not the focus of this thesis.

1.2 Goals and Contributions

As early stated, most previous studies investigated SATD by analyzing TD-related concerns in source code comments [Potdar and Shihab, 2014; Huang et al., 2018; Liu et al., 2018; Sierra et al., 2019a]. The ones that relied on other software artifacts to identify SATD are restricted to primarily exploring indications of TD admission (i.e., investigating whether developers self-admit TD by other means). Particularly, few studies deeply investigated SATD in issue tracker systems, despite the increasing adoption of issues in software development [Cabot et al., 2015]. Therefore, there is still a lack of knowledge about this practice, including the characteristics of the debts documented in issues, as well as the effectiveness of creating and managing issues to reduce TD. In other words, the current knowledge on SATD is mostly restricted to debts documented in a single type of artifact (i.e., most previous results consider only SATD in code comments). Hence, the general objective of this thesis is described as follows:

> We aim to provide an in-depth study on Technical Debt documented using issues (SATD-I), including its characteristics, the interplay with SATD-C, and the circumstances to adopt one or another. We also intend to propose and evaluate a lightweight framework to support developers to manage TD through the creation of issues.

To accomplish this objective, we divided the work in four major working units:

1. First, we explore the characteristics of SATD-I adoption, assessing the types of TD frequently documented in issues, as well as the reasons of its introduction and payment.

2. In the second working unit, we strengthen our initial results by collecting a large-scale dataset of SATD-C and SATD-I items and by investigating the interplay between both forms of SATD.

3. Based on the results of the second working unit—in which we found that SATD-C and SATD-I have distinct natures— we dedicate the third work to investigate the circumstances that drive developers to document SATD either in code comments, or in issues. We also propose a set of guidelines to support developers to choose between one or another.

4. Finally, in the fourth working unit, we propose a framework that considers SATD-I as means to document and manage Technical Debt. The framework includes four major activities concerning TD identification, monitoring, repayment, and documentation.

We also assess its adoption in a real scenario, including two development teams from a large public company.

We summarize each work and highlight their contribution in the remainder of this section.

1.2.1 Characterization of SATD in Issues

Although previous studies have prospected the existence of TD admission in issues excerpts, there is a lack of solid understanding about the adoption of this artifact as means to document Technical Debt. Particularly, we are not sure whether documenting TD in issues is a common practice, as well we are unaware about the characteristics of the TD commonly reported in issues, and the circumstances that drive developers to create and pay such debts. Therefore, in this thesis we initially seek to empirically study and characterize this practice and provide solid insights about SATD-I adoption. In this study, we present the following contributions:

- We identify and study SATD by mining issue tracker systems. We confirm that developers use issues to admit Technical Debt in their projects, i.e., SATD does not appear only in code. For that, we collect and characterize a dataset of 286 SATD-I instances, from five relevant open-source systems.
- We show that almost 60% of the studied SATD-I is related to DESIGN flaws. Other types of SATD-I include UI (10%), TESTS (9%), and PERFORMANCE (8%).
- We also surveyed developers involved in SATD-I payment. As a result, we show that almost 45% of them indicate that SATD-I was introduced as a deliberate choice to ship earlier. Regarding the reasons to pay SATD-I, most of the debts are paid to reduce their interest (65%) or to clean code (28%).

1.2.2 Interplay Between SATD Types

Based on the results of our initial study—in which we show that issues are indeed adopted to document TD—we investigate the interplay between the debts reported in code comments and issues. To accomplish that, we explore the feasibility of implementing a tool to document SATD. Particularly, we evaluate developers interest in transforming

SATD-C instances into GitHub issues, and we also investigate whether it is a common practice to link both forms of SATD. In fact, this tool was suggested by many commenters after our initial study was discussed at Hacker News, in 2020.[2] Therefore, we emphasize the following contributions of this second working unit:

- We strengthen our initial observations by collecting a large-scale dataset of 20K SATD-I issues and 72K SATD-C comments.
- We implement ADMITD: a prototype tool that identifies code comments indicating TD and automatically transforms them in GitHub issues (SATD-I). By applying ADMITD and surveying core developers from 10 GitHub repositories, we show that there is a negligible interest in such transformations.
- We also investigate the viability of tools for linking SATD-C to existing instances of SATD-I. Such tools can make the navigation from SATD-C to SATD-I easier and straightforward. However, we show that linking both forms of SATD is not a common practice (i.e., we found less than 1% of explicit references).
- We conclude by arguing that there is a minor interplay between both forms of SATD, i.e., SATD-C and SATD-I are adopted to report TD in different circumstances.

1.2.3 Guidelines to Document SATD

As we concluded in our second study that SATD-C and SATD-I are adopted to document TD in distinct situations, we dedicate our third study to document the circumstances that drive developers to report SATD either in code comments or issues. Specifically, we elicit a catalog of guidelines to support developers to better document SATD. With this study we achieved the following contributions:

- After surveying 59 developers that documented TD using both SATD-C and SATD-I, we present a catalog of 13 guidelines to better document SATD (six recommendations for choosing SATD-C and seven for SATD-I).
- We show that developers mostly use source code comments to provide context to the reader (58%), to report low priority debts (24%), and to document local-scoped TD (19%).

[2] `https://news.ycombinator.com/item?id=22915584`

- By contrast, issues are used to foster discussion with other team members (31%), to document TD that needs to be tracked (27%), and to report debts that spans in multiple places (25%).

1.2.4 LTD: Less Technical Debt Framework

In the previous studies, we found that (i) issues are indeed adopted to document SATD; (ii) debts documented in code comments and in issues have distinct natures; and (iii) there are specific circumstances where developers find more suitable to adopt one or another. Based on these results, in our fourth effort we explore the effectiveness of adopting issues in activities to manage TD. Particularly, we wrap up the previous knowledge gained with our studies into LTD (Less Technical Debt): a lightweight framework that aims to inject TD concerns in agile-based methodologies. To accomplish that, LTD proposes four activities:

1. TD CONSENSUS: in this activity, the team aims to create a common understanding among developers about TD concepts.
2. TD DISCOVERY: next, the team is stimulated to prospect the TD that already exists in the system and document them through the creation of labeled issues (which we now refer as TD Story).
3. TD PLANING: based on the backlog produced during the TD DISCOVERY, in this activity the team plans the payment of TD in every sprint.
4. TD PAYMENT: finally, in TD PAYMENT days, developers dedicate effort to pay the planned debts.

We also assess LTD in a real-world scenario, involving 30 stakeholders and two development teams (in different development stages) from a large public company. We show that LTD was indeed effective to reduce TD in the studied systems, creating awareness in both teams and stimulating a culture of constantly documenting and paying debts.

1.3 Publications

This thesis encompasses the contributions contained in the following publications:

IEEE Software '22 Xavier, L., Montandon, J. E., and Valente, M. T. Comments or issues: Where to document technical debt? *IEEE Software*, 39(5):84–91, 2022b (**Chapter 5**).

EMSE '22 Xavier, L., Montandon, J. E., Ferreira, F., Brito, R., and Valente, M. T. On the documentation of self-admitted technical debt in issues. *Empirical Software Engineering*, 27(163):1–34, 2022a. (**Chapter 4**).

MSR '20 Xavier, L., Ferreira, F., Brito, R., and Valente, M. T. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020. (**Chapter 3**).

Furthermore, we also contributed to the following work during this Ph.D. research:

EMSE '20 Brito, A., Valente, M. T., Xavier, L., and Hora, A. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25:1458–1492, 2020

SANER '18 Brito, A., Xavier, L., Hora, A., and Valente, M. T. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265, 2018

1.4 Thesis Outline

We organize this thesis as follows:

Chapter 2 covers background information to support this thesis. We provide an overview about Technical Debt, Self-Admitted Technical Debt, and Issue Tracker Systems. We also discuss previous research on SATD, separated in two categories: studies related to SATD-C and studies related to SATD-I. Finally, we highlight the main differences between these works and ours.

Chapter 3 presents our two initial studies aiming to characterize Technical Debt documented in issues. In the first, we present the types of Technical Debt commonly documented in issues. In the second, we unveil the circumstances that drive developers to introduce and pay these debts.

Chapter 4 reports two studies on the interplay between SATD-C and SATD-I. In the first, we investigate developer's interest in automatically creating SATD-I based on SATD-C instances. In the second, we explore the practice of creating explicit references to SATD-I in SATD-C comments.

Chapter 5 unveils a catalog of 13 guidelines to support developers to decide whether to document Technical Debt in comments or issues. For that, we performed a survey with developers responsible for creating both forms of debt.

Chapter 6 presents the Less Technical Debt framework (LTD), describing the four main activities proposed to be injected in agile-based development teams. We also provide the results of an initial case study conducted with two teams from a large public company from Brazil.

Chapter 7 summarizes the conclusions we leveraged throughout this thesis and outlines some ideas we find interesting to investigate in the future.

Chapter 2

Background

We start this chapter by providing an overview about the Technical Debt metaphor and its concepts (Section 2.1). Similarly, we also discuss in Section 2.2 the concept of Self-Admitted Technical Debt and its major fields of study. Then, we present papers that rely on issue tracker systems to collect software engineering artifacts and investigate developers practices (Section 2.3). In Section 2.4, we present the studies that are most directly related to this thesis. Finally, we provide our final remarks in Section 2.5.

2.1 Technical Debt Overview

In 1992, Ward Cunningham resorted to a financial analogy to frame the costs and benefits of sub-optimal implementation decisions in software development [Cunningham, 1992]. Particularly, the Technical Debt (TD) metaphor warns developers about the long-term impact of the quality workarounds commonly adopted to promote earlier releases or rapid software growth. Although the term was quickly embraced by industry, TD began to be investigated in the literature by the early 2000s [Ciolkowski et al., 2021]. In 2010, the first workshop dedicated to the subject promoted the increase of interest in academia. Recently, to consolidate the state-of-the-practice and unify several concepts about TD, researchers and practitioners gathered in 2016 at a Dagstuhl seminar. By the end of the event, it was proposed the following definition of Technical Debt:

> *In software-intensive systems, technical debt is a collection of design or implementation constructs that are expedient in the short term, but set up a technical context that can make future changes more costly or impossible. Technical debt presents an actual or contingent liability whose impact is limited to internal system qualities, primarily maintainability and evolvability.* (Avgeriou et al. [2016])

In comparison to Cunningham's initial proposition, this last definition is not restricted to source code, but also includes concerns related to the whole software development cycle. For example, it considers cases when developers deliver software without tests, neglect documentation updates, or create sub-optimal infrastructure solutions. In this thesis, we follow this modern definition.

A second relevant outcome of this seminar is the conceptual model presented in Figure 2.1. This view summarizes the elements related to Technical Debt. Based on the premise that a system has a set of concerns, the model includes TD as one of them. In this case, the Technical Debt associated to a system is composed by a set of TD Items, which are related to three relevant elements: Cause, Consequence, and DevelopmentArtifact. The first, reflects the circumstances that drive developers to create an item: a process, schedule pressure, or an arbitrary decision. The Consequences associated to a TD item include its impact in terms of the cost of future changes, the value of the system, the schedule, and the quality of the system. Finally, a TD item is also associated to one or more artifacts, such as: the code of the system, its documentation, tests, and defects.

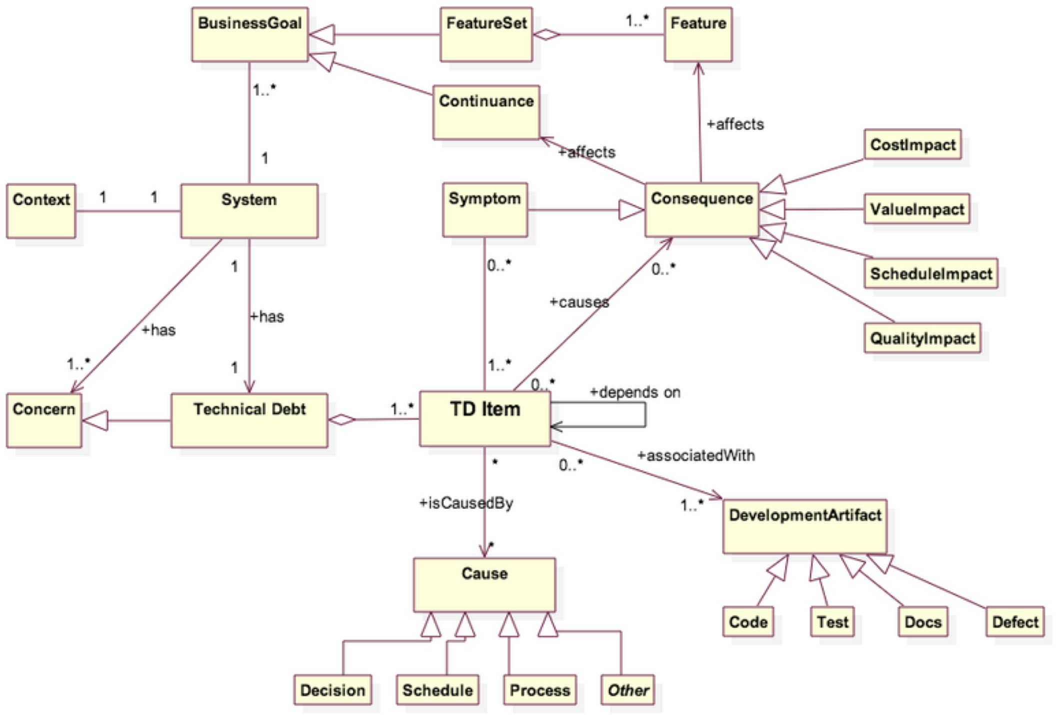


Figure 2.1: Conceptual model for Technical Debt proposed by Avgeriou et al. [2016].

In this thesis, we assume a relationship that is not detailed in the model: a TD Item can be *documented*—or *self-admitted*, as we better discuss in Section 2.2—in one or more artifacts. As illustrated in Figure 2.2, we include a *many to many* association between "TD Item" and a new element called "DocumentationArtifact". These artifacts can be of at least two types: code comments and issues.

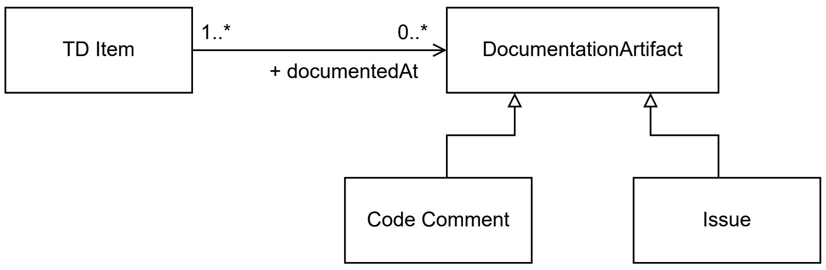


Figure 2.2: Extension of the conceptual model for Technical Debt assumed in this thesis.

After assuming that, in this PhD thesis we conduct an in-depth investigation of the TD items documented using issues, including its characteristics, the interplay with the ones documented in code comments (Chapters 3 and 4), and the circumstances to adopt one or another (Chapter 5). Finally, we propose and evaluate a lightweight framework to support developers to manage TD through the creation of issues (Chapter 6). In the remaining of this chapter, we also discuss two relevant concepts for this thesis: TD categorization (Section 2.1.1), and TD management (Section 2.1.2).

2.1.1 Technical Debt Categorization

Identifying and categorizing a TD Item is useful to promote its payment and prioritization (i.e., different categories of debts usually require different solutions and priorities). For this reason, several authors presented distinct perspectives to organize and classify Technical Debt. In this section, we present two perspectives adopted in this thesis to characterize Self-Admitted Technical Debt documented in issues (Chapter 3). The first refers to the origin and motivations of the debt, resulting in a classification of TD into four categories depicted in the form of a quadrant. The second refers to a set of more specific categories that can be used during the identification, payment, and prevention of TD.

The Technical Debt Quadrants

Firstly, McConnell [2007] divided the Technical Debt into two basic kinds: the ones incurred unintentionally and the ones incurred intentionally. According to the author, debts of Type I (Unintentional Debt) include *"the non-strategic result of doing a poor job"*. For example, a code produced by a novice developer that turns out to be difficult to maintain and evolve. By contrast, debts of Type II (Intentional Debt) occur *"when an organization makes a conscious decision to optimize for the present rather than for the* *future"*. In this case, the author argues that time constraints and release pressure are the most common motivations to provoke such debts.

Extending this initial classification, Fowler [2009] embraced situations when TD is originated by a reckless or prudent posture of developers. In other words, the author considers that TD is originated deliberately or inadvertently, as well as recklessly or prudently. Such categories are represented by the author in the form of a quadrant, as illustrated in Figure 2.3. The four segments in this figure can be explained as follows:

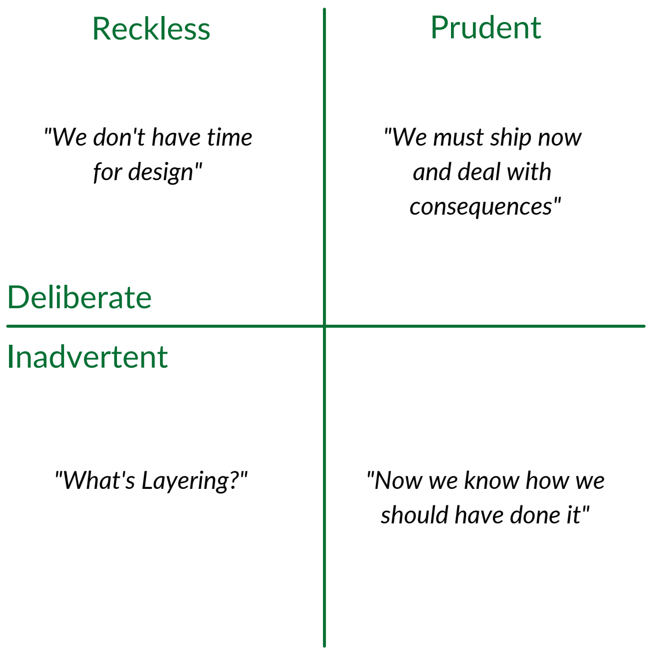

Reckless
Prudent
"We don't have time for design"
"We must ship now and deal with consequences"
Deliberate
Inadvertent
"What's Layering?"
"Now we know how we should have done it"

Figure 2.3: The Technical Debt quadrants proposed by Fowler [2009].

- RECKLESS—DELIBERATE DEBT: this category of TD is originated by the carelessness of developers. In this case, although they know that they are creating a debt, they are not aware about its negative effects.

- RECKLESS—INADVERTENT DEBT: in this type of debt, there is a lack of knowledge on software engineering best practices. In this case, the development team is not aware of the existence of such debts, which can lead to unexpected negative surprises.

- PRUDENT—DELIBERATE DEBT: this category includes TD created on purpose, to reach short-term benefits. Particularly, developers that incur in this type of debt take into consideration the potential long-term consequences. As illustrated in the figure, they assume a posture of *"we must ship now and deal with consequences"*.

- PRUDENT—INADVERTENT DEBT: finally, this type of debt refers to poor solutions latter identified, mostly after the knowledge acquired during the development process. In this case, the debt was originated by developer's lack of knowledge at the time when they were producing the solution.

Specific Types of Technical Debt

Several other authors proposed taxonomies to classify TD according to different aspects [Alves et al., 2014, 2016; Li et al., 2015]. In Chapter 3 of this thesis, we classify the SATD-I in our initial dataset according to the categories proposed by Li et al. [2015]. In this work, the authors describe a systematic mapping to identify and analyze scientific papers on TD from 1992 to 2013. As a result, they propose the following categories:

- DESIGN: refers to technical shortcuts used in internal method design and high-level architecture.
- UI: refers to debt on the elements of user interfaces.
- TESTS: refers to the absence of tests or to workarounds on existing code for testing.
- PERFORMANCE: refers to debt that affects system performance (e.g., in time and memory usage).
- INFRASTRUCTURE: refers to debt on third-party tools, obsolete technologies or deprecated APIs.
- DOCUMENTATION: refers to insufficient, incomplete, or outdated documentation.
- CODE STYLE: refers to code style violations.
- BUILD: refers to debt in building code, as when using scripts that make the build more complex or slow.
- SECURITY: refers to shortcuts that expose system data or compromises user permission access.
- REQUIREMENTS: refers to debt on requirements specification that leads to implementation problems.

2.1.2 Technical Debt Management

As a financial debt—which interest increases over the time—it is crucial that Technical Debt concerns are well managed. Particularly, developers (or other stakeholders) should be aware about the TD Items included in the project, constantly trying to pay the debts, and consequently improving software quality. In most cases, they should find the right balance between keeping postponing the removal of a debt—and accepting its unavoidable increase of interest—or dedicating efforts to pay it. Additionally, deciding which debts should be addressed first is another relevant aspect to be taken into consideration in such scenarios.

In this context, several studies proposed different methodologies to manage TD [Freire et al., 2020; Pérez et al., 2021; Freire et al., 2021; Besker et al., 2019; Eliasson et al., 2015; Rocha et al., 2017; Guo et al., 2016; Rios et al., 2020; Li et al., 2015]. Particularly, they mostly agree on a subset of the activities proposed in the seminal work of Li et al. [2015]. According to the authors, the following activities should be addressed to manage TD:

- Identification: refers to finding TD Items, e.g., by using static code analysis tools.
- Measurement: includes techniques to quantify the amount of TD in a system.
- Prioritization: supports the decision-making process about the debts that should be addressed first.
- Prevention: aims to avoid incurring further debts on top of the already accumulated ones.
- Monitoring: once TD is identified, this activity seeks to keep unpaid debt under observation (in order to avoid letting it go out of control).
- Repayment: refers to actually eliminate the Technical Debt instances from the system.
- Representation/documentation: aims to make TD well-documented and visualized, promoting awareness on the overall situation of the system.
- Communication: promotes an open-space communication among stakeholders, which promotes the discussion of the necessary measures to prevent and pay TD.

The Less Technical Debt (LTD) Framework—proposed in Chapter 6 of this thesis to manage TD through the creation of issues—includes activities that directly promotes Identification, Monitoring, Repayment, and Documentation of TD.

2.2 Self-Admitted Technical Debt

The concept of Self-Admitted Technical Debt (Satd) was first introduced by Potdar and Shihab [2014]. In opposition to a number of previous works—that focused on the detection of TD Items using static analysis tools—the authors observe that developers commonly document TD through source code comments. Through the analysis of more than 100K code comments, they show that (i) 2.4%–31% of source code files document Self-Admitted Technical Debt items, (ii) experienced developers tend to introduce more debts, and (iii) 26%–63% of SATD items gets removed.

Thus, Potdar and Shihab [2014] inaugurated a new branch of study in the Technical Debt field. In this scenario, the relevance of this study emerges not only because it was the first to prospect this strategy to document TD, but also because it contributed with a set of recurrent textual patterns used by developers to this purpose. By manually reading more than 100K code comments, the authors distilled a set of 62 recurring patterns, which include terms like: *hack, fixme, is problematic, this isn't very solid, probably a bug, hope everything will work, fix this crap.* Table 2.1 presents a sample of comments that were identified as indicating SATD.

Table 2.1: Sample of SATD comments identified by Potdar and Shihab [2014].

Project	**Comment**
Eclipse	`// TODO this is such a hack it is silly`
Chromium OS	`// Unsafe; should error.`
ArgoUML	`// FIXME: This is such a gross hack...`
Apache	`/* Ugly, but what else?  */`

Based on these patterns, several studies flourished in the literature. Particularly, Sierra et al. [2019a] mapped the knowledge regarding SATD in three categories:

- DETECTION STUDIES: includes the literature focused on *"proposing, studying or improving approaches, techniques, and tools to identify or detect instances of* SATD*"*. Particularly, these studies advanced the initial approach of Potdar and Shihab [2014] by presenting alternatives that include pattern-based detection, text mining, NLP detection, or change-level detection approaches.

- COMPREHENSION STUDIES: comprises studies that aim to investigate developer's practice of self-admitting TD, and to characterize the life cycle of a SATD. Specifically, studies focused on comprehending SATD seeks to characterize its introduction, evolution, payment, and relation with different aspects of the software process.

- REPAYMENT STUDIES: includes the works focused on approaches, techniques, and tools that seek to remove or mitigate SATD instances.

As we will better discuss in Section 2.4, **most previous studies on SATD relies on source code comments to identify and manage Technical Debt**. We denote this form of SATD as SATD-C. By contrast, in this thesis we follow a less populated branch on the COMPREHENSION STUDIES field, by investigating Self-Admitted Technical

Debt in issues (SATD-I). Particularly, we conduct an in-depth investigation of SATD-I, including its characteristics (Chapter 3), the interplay with SATD-C (Chapter 4), and the circumstances to adopt one or another (Chapter 5). We also propose and evaluate a lightweight framework to support developers to manage TD through the creation of issues (Chapter 6).

2.3 Issue Tracker Systems

Issue Tracker Systems are software systems that maintain a dataset of tickets (or requests) for one or more projects. Such requests are usually related to problems identified by developers, issues, bug reports, or feature enhancements. These systems also provide an interface that allows developers to interact with the reported issues [Bissyandé et al., 2013]. Such interaction may include operations like reporting a new issue, sending an attachment file, or providing comments for an issue [Anvik et al., 2006].

Most open-source projects use a tracking system to support their development process. They provide a central knowledge repository about the issue's handling progress, as well as a communication channel for contributors [Rocha et al., 2016; Anvik et al., 2006]. Currently, there are several Issue Tracker Systems that are used in open-source software development. In this thesis, we mine and analyze issues from the tracking systems associated to GitHub's and GitLab's repositories. In both systems, the reported issues contain elements that provide relevant information about developers activities, such as: issue's title, description, comments, authors, status, opening and closing dates, labels, and closing commits.

In software engineering research, mining Issue Tracker Systems and their associated issues can provide insightful information about developer's activities. This approach has become a commonplace, in which issues are assessed for different purposes. For example, previous studies have collected issues from tracking systems to evaluate tasks related to classification of bug reports [Bettenburg et al., 2008; Camilo et al., 2015; Alonso-Abad et al., 2019]. Other authors assess historical data to identify patterns to assist with automating particular tasks, such as: Rocha et al. [2016]; Choetkiertikul et al. [2015]; Wang et al. [2021]; Kallis et al. [2019]. However, as we will better discuss in Section 2.4.2, few previous authors look at Issue Tracker Systems through the lens of TD.

2.4 Related Work

As a natural consequence of the first studies on Self-Admitted Technical Debt (SATD)—which started with Potdar and Shihab [2014] identifying TD admission in source code comments—most subsequent studies also relied on this kind of artifact to empirically assess SATD. As discussed in Section 2.2, such studies provide insights and conclusions related to detection, comprehension, and repayment concerns. However, in very few cases, different software artifacts were considered as means to document TD. When it comes to studies that relied on issues, previous works prospected this possibility in restricted scenarios and using limited and small datasets [Dai and Kruchten, 2017; Bellomo et al., 2016]. In this section, we discuss the state-of-the-art on SATD, including studies that rely on code comments (Section 2.4.1), and issues (Section 2.4.2). We also include a discussion about studies regarding TD Management (Section 2.4.3). Finally, we conclude by comparing this thesis with previous studies in the literature (Section 2.4.4).

2.4.1 Studies on SATD-C

In order to replicate the findings of Potdar and Shihab [2014], Bavota and Russo [2016] collected a larger dataset of SATD comments, based on the analysis of more than 600K commits and 2 billion comments, from 159 software projects. As a result, the authors first confirm the previous findings. They also observe that the amount of SATD instances increases over time (in different releases) and tends to survive a long time in the system.

Maldonado et al. [2017a] investigate five Java open-source projects with the purpose of examining the amount of TD removed, the developers who performed the removal, the period in which the debt remained in the project, and the activities that lead to its removal. As a result, the authors show that the majority of SATD is removed from projects by the same developer who introduced the debt (i.e., self-removed), as part of bug fixing activities or as part of the addition of new features. Zampetti et al. [2018] perform a follow-up study, based on the dataset elicited by Maldonado et al. [2017a], to quanti- and qualitatively investigate how Self-Admitted Technical Debt is removed. Specifically, the authors assess the amount of SATD removals that are actually accidental transformations, as well as the extent to which SATD removals are documented in commit messages.

Sierra et al. [2019b] investigate the possibility of using source code comments that indicate Technical Debt to resolve architectural divergences. The authors used a dataset of previously classified SATD comments to trace architectural divergences in an open-source system. They found that 14% of divergences could be directly traced. Therefore, they stand that it is viable to use SATD comments as an indicator of architectural divergences. Zampetti et al. [2017] investigated the adoption of SATD-C as a proxy to recommend developers to write new code. The authors also indicated when SATD should be documented (or "self-admitted"). As a result, the proposed approach achieved good results, improving readability, size, and complexity metrics.

Regarding SATD-C identification, Farias et al. [2016], Huang et al. [2018], Liu et al. [2018], and Guo et al. [2021] also identify SATD by mining source code comments. Moreover, other studies propose the use of natural language processing (NLP) techniques to support SATD identification [Maldonado and Shihab, 2015]. For example, Flisar and Podgorelec [2019], Huang et al. [2018], Ren et al. [2019], Fahid et al. [2019], and Wang et al. [2020] use machine learning techniques for automating SATD detection. Maldonado et al. [2017b] proposed a technique to precisely identify SATD, outperforming the current state-of-the-art, based on fixed keywords and phrases. de Freitas Farias et al. [2020] evaluate a set of contextualized patterns to detect SATD-C using code comment analysis. As a result, the authors show that the adoption of pattern-based analysis can contribute to improve existing methods for automatically identifying and classifying SATD items.

Recent research moved in the direction of improving SATD-C identification, removal, and management. For example, Zampetti et al. [2020] showed that SATD removal follows recurrent patterns. They indicated that it is feasible to automatically recommend strategies to pay SATD concerns related to changing API calls, conditionals, method signatures, exception handling, and return statements. Iammarino et al. [2019, 2021] investigated the relationship between refactoring and SATD-C removal. Although, the authors show that refactorings tend to co-occur with SATD-C removals, they highlight that such improvements are part of different activities performed at the same time. Fucci et al. [2020] investigated the extent to which the term "self-admitted" can be used in the context of TD documented in source code. The results suggest that SATD-C may possibly be used as a sub-optimal alternative to perform code review. Kashiwa et al. [2022] investigated the nature of SATD comments introduced during modern code reviews. The authors show that 28%–48% of SATD-C are introduced during code reviews, as such comments are used as means of communication between reviewers and authors.

To explore SATD-C in-depth, several studies focused on particular aspects of this practice. For example, Tan et al. [2021] conducted an online survey to investigate whether practitioners repay their own debt intentionally. As results, the authors highlight the relevance of the sense of self-responsibility in driving developers to repay SATD-C. Maipradit et al. [2020a,b] investigate a particular class of SATD-C, named as "on-hold" SATD. In this case, the authors provide an automated classifier that can identify debts that contain a condition to indicate that a developer is waiting for a certain event or an updated functionality. Fucci et al. [2021] explore developers' habits in SATD annotation. They mainly show that SATD-C related to functional problems or on-hold conditions tend to be more negative. Besides, few SATD comments include external references. Other research include the investigation of SATD-C in particular environments. For example, Azuma et al. [2022] investigate SATD in Dockerfiles. Xiao et al. [2021] characterize SATD-C in build systems. da Fonseca Lage et al. [2019] study usability TD, and Vidoni [2021] explores SATD in R packages.

> In this thesis, we expand the conclusions of previous studies on SATD by evaluating TD documented in issues (SATD-I). We also investigate the interplay between the two forms of documenting TD, i.e., by using code comments and by using issues.

2.4.2 Studies on SATD-I

Early researches indirectly tackled the adoption of issues as means to document Technical Debt. For example, Martini et al. [2018] conducted qualitative studies to investigate the cost of managing TD, the tools used to track it, and how a tracking process is introduced in practice. As a result, they show that only 7.2% of the participants methodically track Technical Debt. In this context, the majority of them adopt issues and backlog tools for this purpose. Yli-Huumo et al. [2016] investigated the practices adopted by eight development teams to manage Technical Debt. For TD documentation, the authors observed that six teams informally adopted the strategy of creating JIRA issues to document TD concerns. As a result, they provide recommendations for adopting the Technical Debt Management activities, including the suggestion of creating issues to document their occurrence. Silva et al. [2016] also investigate the occurrence of TD discussions beyond the code. Particularly, the authors investigate the different types of TD that can lead to the rejection of pull requests. As a result, the authors highlight that design and test debts cause 63% of the rejections.

The first study to directly prospect the idea of analyzing SATD in issues (i.e., SATD-I) was performed by Bellomo et al. [2016]. In this work, the authors manually examined a sample of 1,264 issues, mined from four industry and governmental issue trackers. They found that developers discussed TD in 109 examples. Dai and Kruchten [2017] also studied the possibility of detecting TD in comments of issues by applying natural language processing and machine learning techniques to identify TD in 8,149 issues. As a result, the authors provide 114 keywords that can be used to detect different types of TD from issue descriptions.

Recently, Li et al. [2020] conducted a case study to investigate the existence of SATD in issue discussions. For this, they manually investigated a sample of 500 issues from two open-source projects (Hadoop and Camel). As a result, the authors classified a set of 117 discussions about TD in categories previously proposed in the literature. They also used these discussions to explore identification strategies and payment activities. In a follow-up study, Li et al. [2022] focused on identifying TD evidences in issue trackers. The authors collected a training dataset that includes 4.2K issues. From this dataset, they identified 23K evidences of TD admission. They used this dataset to propose an approach to automatically identify TD evidences in issue trackers. This approach outperforms baseline techniques and indicates that SATD keywords are intuitive.

> In this thesis, we advance the knowledge on SATD by conducting studies that are not restricted to TD documented using code comments. In addition to a first study on SATD-I characteristics, we also investigate the interplay between both forms of documenting TD, i.e., using comments or issues. We also propose and assess in a real and large software organization a lightweight framework to manage TD through the creation of issues.

2.4.3 Studies on TD Management

Managing Technical Debt is relevant, as there is a common understanding about its negative impact in the long-term. In practice, identifying and documenting TD Items should drive developers to reach the ultimate goal of paying debts, and consequently improving software quality. Several experiences in industry demonstrate this particular interest. For example, Haki et al. [2022] report a one-year experience of managing Technical Debt at Credit Suisse (a large and relevant financial company from Switzerland). To help raise awareness about TD, and stimulate actions related to its payment, the authors proposed a digital nudge that was adopted by more than 3,000 teams in the company. The purpose of the nudge was to create a visual overview of the main concerns related to TD in decision-making processes, such as: overall rating, progress circle, evolution trend, etc. As a result, they observed that such approach was effective in reducing TD. The nudge also supported decisions related to software quality and TD management.

In fact, Technical Debt Management is also a well-studied field in the literature. Several previous studies proposed different methodologies to handle TD, regarding prevention strategies [Freire et al., 2020; Pérez et al., 2021; Freire et al., 2021], awareness raise [Besker et al., 2019; Eliasson et al., 2015; Rocha et al., 2017], and project management [Guo et al., 2016; Rios et al., 2020; Li et al., 2015]. For example, Guo et al. [2016] proposed a framework to manage Technical Debt through three general activities: TD Identification, TD Measurement, and TD Monitoring. The central component of the framework is a Technical Debt list, composed by TD items with a set of properties (e.g., its definition, location, and detection date). Ramasubbu and Kemerer [2018] integrated TD management into quality assurance processes. The proposed framework organizes the different processes for Technical Debt Management in three steps: (i) make Technical Debt visible, (ii) perform cost-benefit analysis, and (iii) control Technical Debt.

More recently, Wiese et al. [2022] proposed the TAP framework, which is an acronym for Technical debt Aware Project management. The framework aims to shed light on intentional and unintentional TD, by promoting awareness of the former, and prevention of the latter. Particularly, the authors propose that development teams should create tickets to handle four types of TD-related tasks: Maintenance Ticket, Maintenance Project, TD Tickets, and Deconstruction Tickets. In this context, Maintenance Ticket and Maintenance Project aim to handle short- and long-term unintentional TD, respectively. TD Tickets refer to intentional debts that should be reported by the time the sup-optimal decision is made. Deconstruction Tickets tackle debts that cannot be paid during the project (e.g., legacy code that needs to be kept in parallel). In addition to the creation of TD tickets, the framework also defines: (i) the period when they should be paid (e.g., TD Tickets should be handled before the project ending date); (ii) the person responsible for the ticket (e.g., architect or business analyst).

> In this thesis, we propose and evaluate a framework to manage TD through the creation of issues, called LTD (Less Technical Debt Framework). This framework advances the previous efforts on the field by concentrating efforts on essential activities that can be injected on agile-based methodologies. Besides, it intends to be flexible, customizable, and centered on the creation and management of issues.

2.4.4 Comparison with Previous Studies

As previously stated, most of the studies in the literature focused on a single form of SATD to conduct analysis related to DETECTION, COMPREHENSION, and REPAYMENT, as classified by Sierra et al. [2019a] (see Section 2.2). The majority of these studies focused on SATD-C. To better position this thesis in the state-of-the-art, we compare in Table 2.2 our work with the top-10 most related studies previously published in the literature. For each work, we provide the number of studied SATD-C instances, SATD-I instances, and systems. In two cases, we filled the Satd-C and Satd-I columns with "N/A" because the selected works conducted case studies to assess TD management (i.e., they did not study any Satd instance). We also highlight that none of the studies analyzed the interplay between Satd types (i.e., only our thesis has a "Y" sign in the Interplay column). To provide a better understanding about their focus, we also detail the category of each work.

Table 2.2: Comparison with previous studies.

Study	Satd-C	Satd-I	Syst.	Inter.	Category
This Thesis	72,669	20,265	190	Y	Comprehension, Repayment
Li et al. [2022]	0	4,200	7	N	Detection
Wiese et al. [2022]	N/A	N/A	1	N	Comprehension
Fucci et al. [2021]	1,038	0	10	N	Comprehension
Li et al. [2020]	0	500	2	N	Comprehension, Detection
Maipradit et al. [2020a,b]	335	0	10	N	Comprehension, Detection
Dai and Kruchten [2017]	0	8,149	1	N	Detection
Bellomo et al. [2016]	0	1,264	4	N	Comprehension
Bavota and Russo [2016]	7,584	0	159	N	Comprehension, Detection
Guo et al. [2016]	N/A	N/A	1	N	Comprehension
Potdar and Shihab [2014]	1,263	0	4	N	Comprehension, Detection

2.5 Final Remarks

In this chapter, we started by providing an overview about Technical Debt and its concepts (Section 2.1). Particularly, we presented a modern definition of TD, that is not restricted to debts in source code, and discussed a conceptual model that supports the understanding about the topic. We also detailed two different perspectives to classify TD, as well as the main activities proposed in the literature to manage TD. In Section 2.2, we refined the TD subject, presenting the Self-Admitted Technical Debt field of study. Next, we presented the adoption of Issue Tracker Systems in software development and empirical software engineering (Section 2.3). Finally, we concluded by discussing in Section 2.4 the studies that closely relate with this thesis. We also highlighted in each section the differences between such previous studies and the work described in this thesis.

Chapter 3

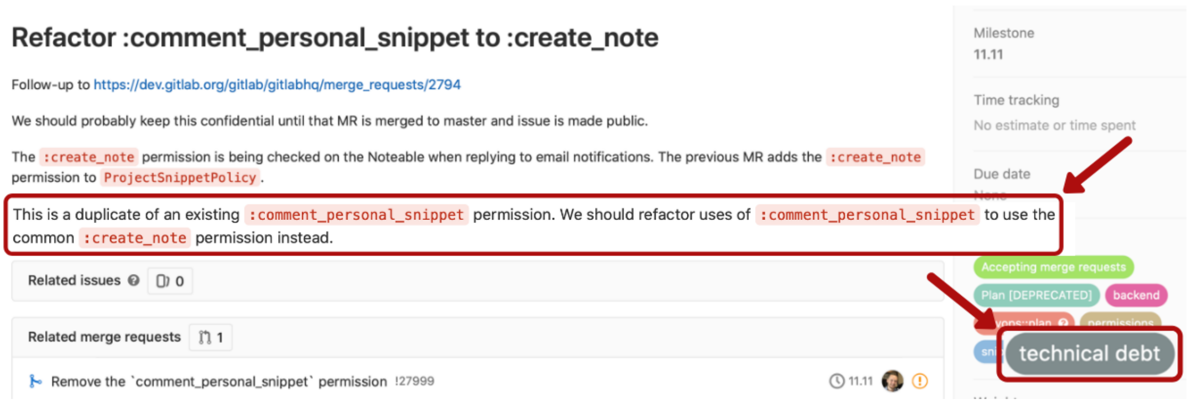
Characterization Study

In this chapter, we perform two exploratory studies to characterize SATD documented in issues. Our goal with these studies is to advance the knowledge about this practice and provide the first insights about SATD-I adoption. In Section 3.1 we introduce and motivate both studies. Next, we dedicate Section 3.2 to present our initial dataset. Section 3.3 presents the classification study performed to assess the types of TD paid in SATD-I. In Section 3.4, we detail the survey conducted to identify the motivations behind SATD-I introduction and payment. Sections 3.5 and 3.6 present implications and threats to validity, respectively. Finally, Section 3.7 concludes the chapter.

3.1 Introduction

Self-Admitted Technical Debt (SATD) is a particular case of TD where developers explicitly admit their sub-optimal implementation decisions [Potdar and Shihab, 2014; Bavota and Russo, 2016; Maldonado et al., 2017a; Zampetti et al., 2018]. To our knowledge, the majority of SATD studies rely on source comments to identify SATD instances. Particularly, they search for specific TD-related terms in source code comments—such as *fixme*, *TODO*, and *hack*. By contrast, in this thesis we argue that **developers can admit Technical Debt out of the source code, by creating issues in tracking systems** documenting their sub-optimal decisions. To document the debt, they label these issues with terms such as *technical debt* or *debt*. An example is presented in Figure 3.1. The figure shows an issue from GitLab requesting the removal of duplicated code (in this case, a permission variable). As we can see, it received a *technical debt* label.

To better characterize SATD documented in issues (SATD-I), we first study its occurrence through the analysis of paid instances, i.e., issues documenting TD that were successfully closed by developers. Our intention is to study SATD-I instances that had a practical and positive impact on the projects. For that, we collect and characterize an initial dataset containing 286 SATD-I instances from five relevant open-source systems,



Refactor :comment_personal_snippet to :create_note

Follow-up to https://dev.gitlab.org/gitlab/gitlabhq/merge_requests/2794

We should probably keep this confidential until that MR is merged to master and issue is made public.

The `:create_note` permission is being checked on the Noteable when replying to email notifications. The previous MR adds the `:create_note` permission to `ProjectSnippetPolicy`.

This is a duplicate of an existing `:comment_personal_snippet` permission. We should refactor uses of `:comment_personal_snippet` to use the common `:create_note` permission instead.

Related issues 0

Related merge requests 1

Remove the `comment_personal_snippet` permission !27999 11.11

Milestone
11.11

Time tracking
No estimate or time spent

Due date
None

Figure 3.1: Example of SATD in a GitLab's issue.

including GitLab (a git-hosting platform that is publicly developed and maintained using its own services), and VS Code (the popular IDE from Microsoft). Developers of these repositories follow a practice to create and label issues that refer to TD problems, i.e., we view these instances as cases of SATD-I. We use this dataset to answer three research questions:

RQ1. What types of Technical Debt are paid in SATD-I? In this RQ, we manually analyze and classify the TD problems documented, discussed, and fixed in the 286 instances of SATD-I from our dataset. To perform this classification, we reuse ten categories of TD from the literature [Li et al., 2015].

RQ2. Why do developers introduce SATD-I? Next, we perform a survey with developers directly involved on SATD-I payment. We analyze 30 received answers (response rate of ∼35%) describing why they introduced the studied SATD-I.

RQ3. Why do developers pay SATD-I? We also elicit a list of five main reasons that drive developers to pay SATD-I by asking the participants of our survey why they decided to close the studied issues. We also shed light on TD interests by investigating the maintenance problems caused by SATD-I.

3.2 Initial Dataset

We first conduct two studies to investigate (i) the types of SATD commonly reported in issues (Section 3.3); and (ii) the motivations that drive developers to introduce and pay such debts (Section 3.4). For these studies, we collect an initial dataset containing SATD-I instances from five open-source systems: GitLab and four GitHub-based systems. We selected GitLab because it is a well-known platform that supports a git-based version control service and also a CI/CD pipeline. Moreover, we had previous knowledge—from our research in the area—on GitLab's practice to label TD-related issues.

In this section, we explain how we selected the GitLab issues used in these initial studies (Section 3.2.1). We also explain how we mined and selected four GitHub projects that follow a practice similar to the one used by GitLab, i.e., they also use specific labels on issues that discuss Technical Debt (Section 3.2.2). Finally, we provide a quantitative overview of this first dataset (Section 3.2.3).

3.2.1 GitLab CE

Differently from GitHub, GitLab's source code is publicly available in the platform, i.e., GitLab is an open-source project that is developed and maintained using its own services. In fact, the project has two editions: Community (CE) and Enterprise (EE). The latter is a commercial version and the former is an open-source edition. GitLab's development happens on both repositories, which are continuously synchronized. Since they are public, we rely on issues from GitLab CE.

First, we used GitLab's REST API to select all issues with a *technical debt* label that were closed in the six months before our search. We only selected closed issues because our primary focus is to explore Technical Debt that was paid. Moreover, we restricted the selection to the last six months to increase the chances of receiving answers in the survey that we performed with GitLab's developers—and also to increase the confidence on the survey answers (see Section 3.4).

After applying the described selection criteria, we found 188 issues. The author of this thesis carefully inspected each one and removed 65 issues (34.6%) that represent duplicated issues, issues that only include discussions, and ignored issues. He also verified that no issue was automatically tagged by a static analysis tool. For example, he discarded an issue where the developer concluded that:

Heh, this is a duplicate of gitlab-ee#3861 (closed), which is being worked on right now by @cablett. I'll close it![1]

Besides, during the classification of the 123 remaining issues, we identified and removed six issues that only request new features, bug corrections, or build failure fixes (i.e., despite having a technical debt label, they are not related with TD). For example,

[1] `https://gitlab.com/gitlab-org/gitlab-ce/issues/34659`

we discarded an issue that reports:

Commit count and other project statistics are incorrect.[2]

After this step, we selected 117 SATD-I instances from GitLab.

3.2.2 GitHub-based Projects

We also searched for SATD-I in open-source GitHub systems. We restricted the search to the top-5,000 most starred GitHub repositories, since stars is a commonly used proxy for the popularity of GitHub repositories [Borges et al., 2016; Silva and Valente, 2018]. We used GitHub's GraphQL API to search for all issues of such repositories that were closed in the previous six months—due to the same reasons explained for GitLab—and that include one of the following labels: *technical debt*, *Technical Debt*, and *debt*. In other words, we decided to select SATD issues by using this set of labels as a precise sign of the presence of such discussions in the issue. As a result, we found 252 issues in 23 repositories. However, we decided to discard 34 issues from 19 repositories with less than 10 issues. The rationale was to focus the study on repositories where labelling issues denoting TD is a common practice.

As previously conducted for GitLab issues, the author of this thesis inspected all 218 issues selected in GitHub (i.e., 252 − 34 issues) and discarded 49 issues (22%) that do not have a clear indication of representing an actual case of TD payment. In the end, 169 SATD-I instances coming from four GitHub repositories were selected for inclusion in our dataset.

3.2.3 Dataset Characterization

Table 3.1 shows the name of the systems in our dataset, the platform where they are hosted (GH refers to GitHub and GL refers to GitLab), the tags they use to denote SATD-I and the number of issues selected in each system. As we can observe, there is a concentration of issues in GitLab-CE (40.9%) and on MICROSOFT/VSCODE (46.2%), which is the popular IDE from Microsoft whose development history is publicly available on GitHub. The remaining SATD-I instances come from INFLUXDATA/INFLUXDB (7.3%),

[2]https://gitlab.com/gitlab-org/gitlab-ce/issues/44726

MIRUMEE/SALEOR (3.5%), and NEXTCLOUD/SERVER (2.1%). INFLUXDATA/INFLUXDB is a platform for time series storage and manipulation. MIRUMEE/SALEOR is an open-source eCommerce platform, and NEXTCLOUD/SERVER is a framework for communicating with Nextcloud (a service for hosting files on the cloud).

Table 3.1: Selected repositories.

Repository	**Plat.**	**Tag**	**SATD-I**	**%**
MICROSOFT/VSCODE	GH	*debt*	132	46.2%
GITLAB/GITLAB-CE	GL	*technical debt*	117	40.9%
INFLUXDATA/INFLUXDB	GH	*Technical Debt*	21	7.3%
MIRUMEE/SALEOR	GH	*technical debt*	10	3.5%
NEXTCLOUD/SERVER	GH	*technical debt*	6	2.1%
Total			286	100%

Figure 3.2 shows violin plots comparing the issues selected in the study with all other issues.[3] We can see that SATD-I takes more time to be closed (16.7 vs 4.0 days, median values). They also have more comments (5 vs 3 comments) and labels (3 vs 2 labels). These observations are statistically confirmed by applying the one-tailed variant of the Mann-Whitney U test (p-value ≤ 0.05). Finally, the last chart shows the code churn of SATD-I versus all issues in our dataset. This metric refers to the number of added and deleted lines of code in the commits responsible for closing the issues. The median code churn is 18 added/deleted lines (paid TD issues) versus 20 added/deleted lines (for all issues). However, in this case, the distributions are not statistically different (p-value = 0.13). i.e., SATD issues are not different from other issues in terms of added and deleted lines of code.

3.3 Classification Study

In this section, we present our first exploratory study. Exploratory research is commonly adopted to investigate a problem that is not clearly defined or studied in the literature. It is conducted to provide a better understanding of the phenomenon and support further conclusions [Wohlin et al., 2012]. In this context, few previous research assessed the usage of issues to admit TD [Cunningham, 1992; Bellomo et al., 2016]. Particularly, none of them used labels as a proxy for TD identification. Therefore, in this

[3]We acknowledge that some of the non-selected issues might also refer to TD (for example, developers may have forgotten to label them as such). Despite that, this fact does not invalidate our key goal in the studies, which is exploring a valid and large sample of issues that explicitly document TD.

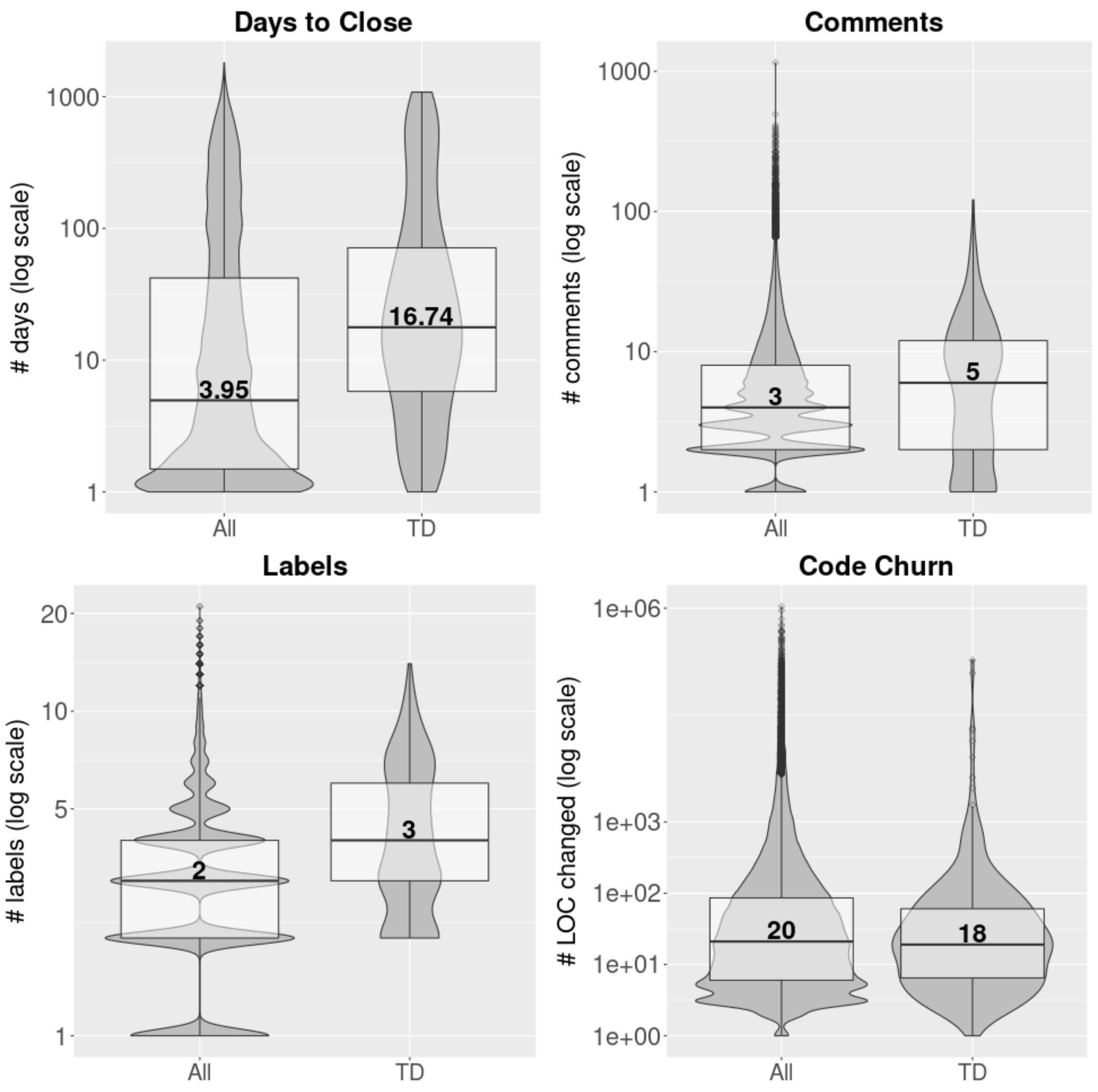


Figure 3.2: Distribution of days, comments, labels, and code churn per issue.

study we aim to explore the types of TD commonly documented, discussed and fixed using labeled issues. Particularly, we seek to answer our first research question:

RQ1. What types of Technical Debt are paid in SATD-I? We aim to reveal the main types of TD documented in SATD-I. Similar classifications have been previously performed in the literature for SATD-C [Maldonado and Shihab, 2015]. In this RQ, we seek to expand this knowledge by considering SATD types beyond the code.

We dedicate Section 3.3.1 to present the methodology applied to classify the SATD-I instances in our dataset. In Section 3.3.2, we present the results as follows: first, we discuss issues related to DESIGN (the most popular type of TD) and its corresponding sub-classification; next, we present the results for the other types of SATD-I.

3.3.1 Study Design

To identify the types of Technical Debt paid by developers, we carefully analyzed 286 SATD-I instances using *closed-card sort* [Spencer, 2009], a technique to classify a set of documents into predetermined categories. This technique involves the following steps: (i) defining the set of categories, (ii) initial reading of the issues, (iii) classifying the issues by independent researchers, (iv) resolving conflicts. We perform closed card sorting using categories previously elicited in the literature. Specifically, we reused the categories described in a study performed by Li et al. [2015]. As described in Section 2.1.1, the authors conducted a systematic mapping study, providing a taxonomy of TD types that includes: DESIGN, UI, TESTS, PERFORMANCE, INFRASTRUCTURE, DOCUMENTATION, CODE STYLE, BUILD, SECURITY, and REQUIREMENTS.[4]

Since DESIGN was the most popular case of SATD-I (as we found in our first round of classification), we decided to perform a sub-classification of this type of issue. Thus, we defined four categories:

- COMPLEX CODE: refers to intra-method poorly implemented code or to naming issues.
- ARCHITECTURE: refers to high-level design problems, including inadequate organization of packages.
- CLEAN UP: refers to the elimination of obsolete or dead code.
- CODE DUPLICATION: refers to code clones that should be removed to improve maintainability.

After defining the mentioned categories, the author of this thesis and two research collaborators manually analyzed the issues, by reading their descriptions and existing discussions, with the goal of assigning one (or more) categories. Each issue was analyzed by two independent researchers. In 178 cases (62.2%) they agreed in the first proposed classification. For the DESIGN subclassification (169 issues), the researchers agreed in 92 cases (54.4%). The key challenge in this classification was to understand the purpose of the issue, based on the different pieces of information that it includes (i.e., title, body, comments, labels and closing pull/merge-requests). Therefore, there were cases in which the researchers prioritized different aspects to classify the issue, resulting in conflicting results. These cases were solved during the final step of the *closed-card sort* technique,

[4] Although this taxonomy is based on research published before the SATD term was coined, we decided to use it because it covers problems that can occur beyond the source code (e.g., UI and BUILD problems).

where the researchers discussed each conflict and reached a consensual classification considering all aspects of the issue. To facilitate the identification and discussion of the issues in this section, we label them using the initials of the repository name (i.e., VS refers to MICROSOFT/VSCODE; GL to GITLAB/GITLAB-CE; IF to INFLUXDATA/INFLUXDB; SL to MIRUMEE/SALEOR; and NX to NEXTCLOUD/SERVER). The initials are then followed by an integer ID (e.g., GL43 refers to issue 43 from GitLab).

3.3.2 What types of Technical Debt are paid in SATD-I?

SATD-I related to Design

With 169 occurrences (59.1%), most of the selected issues refer to DESIGN debt. In this case, we classified DESIGN SATD-I into four subcategories, as presented in Table 3.2. As we can see, COMPLEX CODE is the type of DESIGN TD more commonly paid by developers (43.8%), followed by ARCHITECTURE (33.7%), CLEAN UP (18.9%), and CODE DUPLICATION (3.5%). Next, we describe and provide examples for each DESIGN category.

Table 3.2: Design SATD-I Classification.

Technical Debt	**Occ.**	**%**
COMPLEX CODE	74	43.8%
ARCHITECTURE	57	33.7%
CLEAN UP	32	18.9%
CODE DUPLICATION	6	3.5%

Complex Code. In 74 cases (43.8%), DESIGN issues are related to technical shortcuts that developers take when implementing methods. In this case, the payment involves changes only in the single method where the debt is located. As an example, the following issues are related to this type of DESIGN SATD-I:

We should unify naming related to checkout functionality, as currently, we're mixing "checkout" with "cart", which leads to confusion when reading the code. I recommend that we settle on the name "checkout" and rename the Cart model and all other occurrences of cart. (SL1)

Currently, errors is an optional list of optional errors. While returning an empty list is probably not needed, current type forces the client to make sure the errors themselves are not null. (SL10)

Architecture. With 57 occurrences (33.7%), the second type of Design TD most commonly paid by developers is related to high-level design flaws. To pay this type of debt, it is usually necessary to make changes in the organization of packages and modules, for example. The following issues are related to this type of Satd-I:

This class is way too big for its own good. For example, there's no need for it to update a project's main language in the same job/thread/process as the other work. (GL43)

The root of the TimeMachine tree contains a TimeSeries component. This component handles fetching time series data used in the TimeMachine (...) The aim of this refactor would be to move all state from the TimeSeries component into Redux and all logic into a thunk. (IF12)

Clean Up. Next, issues related to the presence of obsolete or dead code represent the third most common type of Satd-I, with 32 instances (18.9%). As an example, the following issue is related to this type of Design TD:

In Milestone 11.4, we introduced personal_access_tokens. token_digest, so we can now remove personal_access _tokens.token. (GL47)

Code Duplication. Finally, with 6 occurrences (3.5%), the least common type of Design Satd-I refers to duplicated code, as illustrated by the following issue:

There has been a lot of duplication of frontend code between Protected Branches and Protected Tag feature, this issue is intended to reduce duplication. (GL81)

Other Types of SATD-I

Table 3.3 presents the classification of the remaining Satd issues. As we can see, if we do not count issues related to Design (59.1%), the most common type of paid TD refers to UI (10.1%), Tests (8.7%), and Performance (8%). Next, we describe these categories.

Table 3.3: Other Types of Satd-I.

Technical Debt	**Occ.**	**%**
UI	29	10.1%
Tests	25	8.7%
Performance	23	8%
Infrastructure	18	6.3%
Documentation	12	4.2%
Code Style	8	2.8%
Build	4	1.4%
Security	3	1.1%
Requirements	3	1.1%

UI. With 29 occurrences (10.1%), the second most common type of Satd-related issues refers to debt on user interface code. In this case, developers implemented shortcuts that result in usability flaws, as mentioned in the following issue:

Today there is a "Building..." label appearing around the problems entry when building a project. I think this originates from a time where we did not have support to show progress in the status bar. (VS29)

Tests. In 25 cases (8.7%), Satd-related issues report technical debt concerned to tests. This type of debt is mostly related to: (i) the absence of tests; or (ii) the suboptimal implementation of existing tests. Particularly, the latter refers to automated tests poorly implemented (e.g., tests that need to be refactored to include mocks or to avoid direct access to API servers). The following issue illustrates this type of TD:

I want to have some tests that will give me a better perspective for usage of DB queries under GraphQL API. I would like to have explicit logic to validate that it works as expected. (SL3)

Performance. With 23 occurrences (8%), the fourth most common type of Satd-I is related to performance concerns, in terms of time or memory usage. This is illustrated as follows:

underscore.js is bundled in vendor/core.js but it's the unminified version. Can we replace it with the minified version? The file size is a lot smaller. (NX1)

Every widget and actions in each extension has a global listener to check if there is a change and update itself. This causes 100s of listeners being added to a global event. (VS65)

> Satd-I is paid mostly to fix Design flaws (∼60%). But we also found paid TD related to UI (10%), Tests (9%), and Performance (8%), for example.

Multiple-Category Types of SATD-I

After concluding this classification study, a research collaborator involved in the card sorting activity reanalyzed all TD issues classified as UI, Performance, Build, and Infrastructure. The goal was to check whether these issues also include a discussion related to design or architecture concerns. The results are summarized in Table 3.4. As can be seen, multiple-category discussions are not common. In fact, they occurred in only two categories.

Table 3.4: Multiple-Category Discussion in SATD-I.

Technical Debt	# Instances	# Multiple-Category
UI	29	2
PERFORMANCE	23	2
INFRASTRUCTURE	18	0
BUILD	4	0

3.4 Survey with Developers

In the initial part of our study, we also conducted a survey to reveal developers motivations for SATD-I insertion and payment. Particularly, we surveyed developers responsible for closing the 286 SATD-I instances collected in our first dataset. We relied on their responses to answer the following research questions:

RQ2. Why do developers introduce SATD-I? In this RQ, we reuse Martin Fowler's quadrant [Fowler, 2009] to understand the origin of the studied SATD-I instances (i.e., TD latter admitted vs TD introduced to ship earlier).

RQ3. Why do developers pay SATD-I? Next, we move a step further to investigate the motivations that drive developers to pay SATD-I. Besides, we also elicit the main problems caused by these debts.

We first present the methodology followed in this survey (Section 3.4.1). After that, we present the results for each research question (Sections 3.4.2 and 3.4.3).

3.4.1 Survey Design

To conduct this survey, we sent emails to developers that closed the SATD issues studied in this chapter. Specifically, we selected from our initial dataset developers with public email address who were responsible for (i) closing a specific issue; or (ii) accepting a pull/merge-request that closes the issue (in GitLab, merge-requests are equivalent to pull-requests). From the total of 286 issues, we retrieved a list of 85 distinct emails. In the cases where the same developer was responsible for more than one issue, we selected the most recently closed one.

For each developer, we sent the questionnaire in an interval of at most six months after the date when the issues were closed. Figure 3.3 shows the template of the survey

I figured out that you closed the following issue from [repository name]:

[issue title] [issue link]

which is labeled as [TD-related tag].

I kindly ask you to answer the following questions:

1. Why did you decide to pay this TD?

2. Could you describe the maintenance problems caused by this TD?

3. Could you classify this TD under the following categories:

a. It was deliberately introduced to ship earlier
b. When it was introduced, we were not aware about the best design
c. Other answers (please clarify)

Figure 3.3: Email sent to developers who paid Satd-I.

email. First, we presented the issue that represents the debt paid by the developer. Next, we proposed three questions with the goal of (1) investigating the reasons why developers pay Technical Debt; (2) unveiling maintenance problems caused by TD; and (3) understanding the intentions behind TD insertion. Questions (1) and (2) were open-ended, while question (3) provided two predefined options, reused from the Technical Debt quadrant proposed by Fowler [2009]. According to the author, Technical Debt can be classified into a quadrant divided into reckless/prudent and deliberate/inadvertent debt. We decided to give options only related to the deliberate/inadvertent axis. The rationale is that the reckless/prudent classification might result in biased answers because it requires the developer to make a self-judgment of his/her own work (i.e., he/she needs to classify his/her own work as reckless or prudent). Although developers could simply select one of the answers, we allowed them to provide their own answers or to include comments to predefined answers.

We received 30 answers coming from developers of four repositories (i.e., response rate of 35.3%). Table 3.5 details the number of emails sent and the answers received per repository. microsoft/ vscode and influxdata/influxdb have the highest response rate (both with 40%). However, they do not represent the majority of the answers, once we received 23 answers from GitLab developers.[5]

To interpret the survey answers (1) and (2), the author of this thesis followed an

[5]It is worth mentioning that 33 developers (out of the 85 contacted) were also responsible for opening the corresponding issues. Among the 30 developers who answered our emails, 10 were also the authors of the Satd-I.

Table 3.5: Survey answers.

Repository	**Sent**	**Answers**	**%**
MICROSOFT/VSCODE	10	4	40%
INFLUXDATA/INFLUXDB	5	2	40%
GITLAB/GITLAB-CE	66	23	34.9%
NEXTCLOUD/SERVER	3	1	33.3%
MIRUMEE/SALEOR	1	0	0%
Total	85	30	35.3%

open card-sorting [Spencer, 2009]. This technique is used to identify themes (i.e., patterns) in textual documents through the following steps: (i) identifying themes from the answers, (ii) reviewing the themes to find opportunities for merging, and (iii) defining and naming the final themes. During the analysis, one answer was discarded because the developer did not actually discuss the issue. In a final step, a research collaborator reviewed and confirmed the proposed themes. In the following discussion, we label the quotes with D1 to D29 to indicate developer's answers.

3.4.2 Why do developers introduce SATD-I?

To answer this question, we provided two predefined options: the first is related to developer's decision of introducing TD as a choice for agility. The second corresponds to the scenario where developers only perceived the debt after it was introduced. We also left the opportunity for developers to clarify their answers and provide further information. Figure 3.4 presents the obtained results. As we can observe, most of the studied debts were intentionally introduced by developers *to ship earlier* (12 answers). In nine cases, developers were not aware of the TD when it was introduced (i.e., the debt was initially unintentional and then *later admitted*). Finally, six developers provided other motivations. Next, we detail each of these reasons and provide quotes from extra comments discussed by developers.

SATD introduced to ship earlier. In 12 answers (44.5%), developers confirmed that the Technical Debt was introduced to speed up development. In other words, to deliver faster, developers consciously added shortcuts in their code which were expected to be fixed in the future. To remind about this fact, they also decided to document the TD using an issue. D7, D16, and D9 provided further details for this reason:

It was thoroughly discussed and weighed up before we take the decision to accept the TD

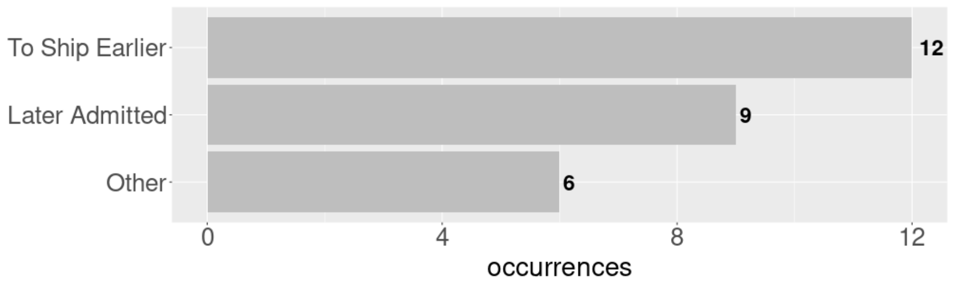


Figure 3.4: Reasons for introducing SATD-I.

to be dealt with on a next release. The TD *wasn't introducing any critical performance issues or bugs to the system. Furthermore, we were confident that we could fix the* TD *in the next release, which happened.* (D7)

I think that usually when we introduce a technical debt it either helps us to ship something earlier/faster or makes first iteration of implementation much easier in general. (D16)

We were aware and were ok with the implementation for now as long as we fixed it afterwards (D9)

Later admitted TD. For nine developers (33.3%), the debt was originated by their lack of understanding about the best design solution at the time the code was initially implemented. After discovering or facing the TD, they decided to admit it opening an issue. The following answers illustrate this scenario:

We figured we'd never hit "that" usecase. But we did. (D23)

The class just grew over time without planning. (D15)

Other reasons. Finally, six developers provided other reasons for introducing TD in their code (22.2%). Answers include the advent of new technologies that turned the old code a debt, and also the mischoice of design alternatives. This is illustrated in the following examples:

It slowly became TD *while at the time of the initial development it was most likely fine to code that way.* (D29)

I think the original author just overlooked that exposing these methods wasn't really needed. (D13)

In most of the cases, SATD-I is introduced as a deliberate choice for agility (44.5%).

3.4.3 Why do developers pay SATD-I?

In order to investigate the reasons why developers pay SATD-I, we combine answers from questions (1) and (2) of our e-mails (Section 3.4.1). First, we directly asked developers the reasons that drive such payment. Next, we complement our findings by eliciting a list of associated maintenance problems.

We first identified five distinct reasons why developers pay SATD-I, as reported in Table 3.6. As we can see, reducing TD interest is the most common motivation for SATD-I payment (65.5%), followed by the desire to have a clean code (27.6%). In some cases, a given answer produced more than one motivation. This explains why the number of occurrences is higher than the number of answers (29 answers). Next, we discuss these reasons.

Table 3.6: Reasons why developers pay SATD-I.

Why did you decide to pay this TD?	**Occ.**	**%**
To reduce TD *interest*	19	65.5%
To clean code	8	27.6%
To get familiarised with the codebase	2	6.9%
To collocate with other related work	2	6.9%
To increase test coverage	1	3.5%

To reduce TD interest. With 19 answers (65.5%), the most common reason for paying SATD-I is to reduce TD interest. Although developers did not directly mention the term *interest*, eliminating the maintenance burden caused by the studied issues was mentioned in several answers. For example, D1 and D19 mention this motivation:

This was adding extra maintenance for me. (D1)

The component was growing too big, making it difficult to maintain. (D19)

To clean code. In eight cases (27.6%), Technical Debt payment is related to the desire of having a clean codebase (e.g., to reduce code complexity and remove duplication). For example, the following answers are related to this motivation:

To keep the code clean and easy to read/maintain. (D27)

To get the benefits of a cleaner code (...). After fixing the TD, *understanding the code got easier. It also got smaller.* (D7)

Technical Debt is periodically paid to reduce its interests (66%), and to clean code (28%).

We also asked the survey participants to comment on the specific maintenance problems that motivated them to close the studied SATD-I instances. Table 3.7 presents the list of the most common answers. In this case, three answers (out of the 29 analyzed) were discarded because they were not clear. According to the remaining answers, TD is mostly responsible for slowing down code evolution, increasing maintenance effort due to duplicated code, and making it harder to read and understand code. The three problems occur with the same frequency (six answers for each).

Table 3.7: Maintenance problems caused by Technical Debt.

What problems are caused by this TD?	**Occ.**	**%**
Code was difficult to evolve	6	20.7%
Duplicated code was demanding extra effort	6	20.7%
Code was difficult to read and understand	6	20.7%
Code performance was poor	5	17.2%
Code was error-prone	4	13.8%
UI presented visual defects	1	3.5%

Technical Debt is commonly responsible for slowing down code evolution, duplicating maintenance effort, and making it harder to read and understand code.

3.5 Implications

Based on the results presented in this chapter, in this section we highlight implications on tool support and process improvement.

Tool Support. In this chapter, **we confirmed that Technical Debt is also documented using issues and we characterized this practice in a set of well-known open-source projects.** Particularly, in RQ2 developers point that the majority of SATD-I instances are intentionally created to ship earlier. Therefore, these results reinforce that developers usually decide to follow the “done is better than perfect” maxim, implementing suboptimal code solutions in order to deliver on time. To tackle this problem, we envision research on new tools to allow developers to explicitly label new debts inserted on GitHub/GitLab-based systems. Specifically, such tools would be responsible for asking pull/merge-requests authors whether the contribution contains any type of TD. In the cases when they admit it, the tool would suggest the automatic opening of a follow-up issue, tagged with a SATD-related label (as in the issues we studied in this chapter). These tools would be effective for various roles of contributors, since: (i) core developers would benefit from managing code quality and better reviewing contributions, (ii) pull/merge-request authors would feel responsible for their own debts (and possibly will come back to pay them), and (iii) newcomers could pay these issues as a way to get familiarized with the code (as we will better discuss in the next implication). Moreover, the number of open TD-related issues could be used as a metric to measure the quality of the system. Although there are several approaches to automatically identify TD on source code (e.g., Maldonado et al. [2017b]; Huang et al. [2018]; Liu et al. [2018]), we claim this tool is based on developers' feedback right after TD insertion. It would also be independent of programming language.

Process Improvement. Among the reasons that drive developers to pay Technical Debt (elicited in RQ3), we identified two motivations explicitly related to software development process: *to reduce TD interest* (with 65.5% of occurrences) and *to get familiarized with the codebase* (with 7%). In other words, this result shows that paying Technical Debt represents an actual activity introduced in the development process of the studied projects to preserve internal quality and to train new contributors on the structure of the code. Therefore, we extrapolate this finding by suggesting two particular implications:

1. We suggest the formalization of Technical Debt payment as an actual activity on modern development processes. Indeed, in Chapter 6 we propose the Less Technical Debt framework (LTD). The general purpose of LTD is to introduce TD-related concerns into agile-based methodologies—in a non-invasive and customizable way—supporting development teams to better manage and pay TD. To accomplish that, the framework proposes the injection of four distinct activities: TD CONSENSUS, TD DISCOVERY, TD PLANNING, and TD PAYMENT, which are better described in Section 6.2.

2. We also advocate that paying Technical Debt may be included on onboarding activities for new team members [Steinmacher et al., 2016]. In this case, team leaders would *"delegate this work to newcomers to give them easy stuff to familiarize themselves with the work process"* (D19).

3.6 Threats to Validity

We acknowledge that the studies reported in this chapter are restricted to 286 closed issues classified as Technical Debt according to SATD-related labels. Although the issues were selected from relevant repositories, maintained by organizations like Microsoft and GitLab, we cannot generalize our findings to other systems, especially to the ones that apply different approaches to manage Technical Debt (i.e., do not use TD-related labels). Moreover, we selected the issues in our initial dataset by using TD-related labels as a proxy for Technical Debt identification. However, as discussed by Kruchten et al. [2012], the concept of Technical Debt has been diluted since its original proposition. Thus, the misunderstanding of this concept by those who added the TD-labeled issues would affect the results of our study. To alleviate this threat, the author of this thesis carefully analyzed the initial selection of 406 TD-labeled issues, and discarded 120 issues (29.6%) that did not have a clear indication of TD payment.

Regarding the classification study reported in Section 3.3, we should mention the subjective nature of the closed-card sort method. Despite the rigor followed by the researchers to perform the classification, the replication of this activity may lead to different results. To mitigate this threat, special attention was paid during the discussions to resolve conflicts and to assign the final themes.

We also acknowledge that the results presented in our second study (Section 3.4) are based on the opinion of 29 developers, mostly from GitLab. Despite that, we claim that the obtained response rate (35.3%) represents a relevant mark in typical software engineering studies. Finally, against our belief, the correctness of developers answers is also a threat to be reported. To alleviate it, we restricted our study to issues closed in the last six months, which was important to guarantee a higher response rate and to increase answers reliability.

3.7 Final Remarks

In this chapter, we performed two exploratory studies with the purpose of answering three research questions on Self-Admitted Technical Debt documented through labelled issues (SATD-I). We analyzed 286 SATD-I instances to (1) identify the types of SATD-I more frequently paid, (2) understand the intentions behind SATD-I insertion, and (3) investigate the reasons why developers pay SATD-I. As a result, we show that:

- In almost 60% of the cases, we found that SATD-I is related to DESIGN flaws (with a concentration on method-level debt in 44% of this total);
- About 45% of the studied debt was introduced to ship earlier;
- Most developers pay SATD-I to reduce its interest, and to have a clean code.

Replication Package. We provide the complete dataset used in this chapter and a replication package at: `https://doi.org/10.5281/zenodo.5119967`.

Chapter 4

Interplay between SATD Types

In this chapter, we investigate the interplay between TD documented in source code comments (SATD-C) and issues (SATD-I). Particularly, we collect and explore a large-scale dataset—including instances of both SATD types—to asses whether there is an overlap between both strategies. We dedicate Section 4.1 to introduce and motivate the chapter. In Section 4.2, we detail our new large-scale dataset. Next, in Section 4.3, we present AdmiTD tool and asses the interest of developers to create issues from SATD-C. Section 4.4 explores the links between SATD-C and SATD-I. In Sections 4.5 and 4.6, we provide implications and threats to validity, respectively. Finally, Section 4.7 concludes the chapter.

4.1 Introduction

In Chapter 3, we first confirmed that Technical Debt is also admitted using issues and then we characterized this practice in a set of well-known open-source projects. However, **there is still a lack of knowledge on the interplay between SATD-C and SATD-I**. Particularly, it is not clear whether developers adopt only one of these approaches to admit their debts or whether there is an overlap between them. In this chapter, we tackle this problem by investigating the adoption of tools to support developers in documenting Technical Debt using issues, based on the occurrence of SATD-C instances. i.e., we investigate whether SATD-C comments refer or can be converted to SATD-I instances. Specifically, we evaluate the feasibility of tools to deal with SATD in two distinct aspects:

- To automatically create issues to document TD concerns expressed in source code comments.
- To automatically create links between SATD-C and SATD-I instances that are related.

This tool support was suggested by many commenters after our initial study (Chapter 3) was indexed by Hacker News, in 2020.[1] Indeed, a search on GitHub shows that open-source developers are already working on similar tools, particularly on tools of type (a), as labelled before. As examples, we have tools such as GITHUB-TODOS[2] (1.3K stars), TODO[BOT][3] (701 stars) and TODO TO ISSUE ACTION[4] (211 stars). Our key motivation is to show whether such tools are effectively useful.

To accomplish that, we first create a large-scale dataset of 20,265 SATD-I instances and 72,669 SATD-C instances, extracted from 190 GitHub projects. We use this dataset to answer two research questions:

RQ4. Are developers interested in tools to create issues from SATD-C? First, we implement a prototype tool for automatically generating issues from SATD comments as GitHub issues, called ADMITD. We equipped this tool with a set of heuristics to enhance the acceptance of its recommendations. Then, we validate our results with the principal developers of ten open-source projects.

RQ5. Do developers refer to SATD-I in SATD-C? Next, we search for explicit links between SATD comments and SATD issues. Specifically, we intend to check whether developers mention SATD-I IDs or URLs in SATD-C instances, such as in following code comment from COCKROACHDB/COCKROACH:

```
// TODO(irfansharif):  We should reconsider usage of
NodeLivenessStatus.  (...)  See #50707 for more details.
```

The existence of such links would mean that SATD-C and SATD-I are somehow related (at least in some cases). As a consequence, it would be feasible to implement tools that detect related SATD-C and SATD-I instances, although the SATD-C does not include a reference to the associated SATD-I. In such cases, the tool could recommend the creation of the link.

4.2 Extended Dataset

To perform this extended study, we first build a new dataset, including both SATD-C and SATD-I instances. We decided to replace our initial dataset for three reasons:

1. To include more systems (5 systems vs 190 systems in this new dataset);

[1]`https://news.ycombinator.com/item?id=22915584`
[2]`https://github.com/naholyr/github-todos`
[3]`https://github.com/JasonEtco/todo`
[4]`https://github.com/alstr/todo-to-issue-action`

2. To include Satd-C instances (the initial dataset covered only Satd-I instances);
3. To also consider opened issues, since they might also benefit from the kind of tool investigated in this section.

As a result, this new dataset includes 20,265 Satd-I instances and 72,669 Satd-C instances from 190 GitHub projects. We explain the mining steps that we followed to select the new repositories and their Satd instances in Section 4.2.1. Next, we provide an overview of the new dataset in Section 4.2.2.

4.2.1 Mining Steps

To build this new dataset, we performed the following steps:

1. Project selection. We first focused on retrieving repositories in which Satd-I is a common practice (since it is the least explored form of Satd). As in Section 3.2, we searched for TD-related labels among the top-5,000 GitHub repositories, ordered by number of stars [Silva and Valente, 2018; Borges et al., 2016]. In contrast to our initial dataset—where we selected repositories containing three specific labels—we now adopted different steps to increase the number of retrieved repositories. We started this procedure by identifying the labels related to TD among all the 97,106 labels adopted in the 5K most-popular GitHub repositories. From this initial amount, we removed labels associated with less than 10 issues. The rationale is to discard labels that are not frequently used. By applying this filter, we discarded 60,032 labels, such as: *today*, *announcement*, and *Partner*. After that, we adopted multiple regular expressions to remove commonly used labels that *do not* denote TD [Cabot et al., 2015]. For example, we removed labels like *bug* (2,635 labels), *enhancement* (1,828 labels), *feature* (1,606 labels), and *question* (1,455). By applying these heuristics, we discarded 19,209 labels. Finally, the author of this thesis manually read the remaining 17,865 labels ($97,106 - 60,032 - 19,209$), in order to select the ones that explicitly indicate Technical Debt. e.g., *tech debt*, *debt*, *cleanup*, *workaround*. A research collaborator also inspected a set of 500 randomly selected labels to confirm the classification. As a result, we obtained 219 labels, related to 190 repositories.

2. Mining Satd-I instances. Based on the initial selection of 190 repositories, we used GitHub's GraphQL API to search for issues tagged with the defined Satd-related labels. i.e., we used the set of 219 labels collected in Step 1 as an indication of TD concerns documented in the studied issues. We also considered both open and closed issues. The rationale is to investigate both existing and paid instances of TD. As a result, we selected 20,265 Satd-I instances (4,866 open issues and 15,399 closed issues).[5]

3. Identifying Satd-C instances. Finally, we collected Satd-C instances by cloning and parsing the source code of the 190 repositories selected in Step 1. We analyzed the latest version of each repository at the moment of cloning. For each file, we first extracted code comments by using a regex-based service provided by the Python API *comment_parser*.[6] Then, we filtered every comment containing at least one of the following terms: *TODO*, *workaround*, *fixme*, and *hack*. We selected these terms as they are the most popular ones when reporting TD instances in source code comments [Huang et al., 2018; Potdar and Shihab, 2014; Bavota and Russo, 2016]. As a result, we identified 74,306 comments, distributed through 182 repositories, i.e., in eight repositories, no comment including such terms were identified. To validate this selection, the author of this thesis inspected 3K comments (randomly selected). He confirmed all of them indeed refer to Satd-C. However, in order to have an additional opinion, a research collaborator analyzed a subset of 500 comments, also randomly selected from the initial sample of 3K comments. He also confirmed all of them are Satd-C instances. Next, a second research collaborator inspected the same subset of 500 comments. During the analysis, he raised a discussion about Satd-C instances that do not include any description about the debt (e.g., comments that only include a TODO without any further textual description). As a result, we decided to remove 1,592 comments that not include text besides the searched words. During this validation, we found a bug (or a limitation) in our regex tool when handling very large comments. Therefore, we decided to remove comments with more than 64K characters (i.e., 45 comments, in total). We also inspected some of these removed comments and found they are indeed false positives (all of them refer to minified JavaScript files, for example). In the end, our extended dataset includes 72,669 Satd-C instances (i.e., $74,306 - 1,592 - 45$).

4.2.2 Dataset Characterization

Our extended dataset includes 20,265 Satd-I and 72,669 Satd-C instances, collected from 190 well-known GitHub repositories (10,075 stars per repository, on the median). For example, the list of studied repositories includes: vuejs/vue (186K stars),

[5]We acknowledge that some of the selected issues might not document TD concerns (for example, developers may have incorrectly labeled some issues). Despite that, this fact does not invalidate our key goal in this study, which is studying tool support for Satd documentation.

[6]`https://pypi.org/project/comment-parser/`

MICROSOFT/VSCODE (114K stars), KUBERNETES/ KUBERNETES (79K stars) and ANGULAR/ANGULAR (75K stars). Figure 4.1 describes the distribution of both SATD forms in such repositories, in terms of absolute and relative numbers. We can observe a concentration on code comments as means to document SATD (i.e., 81 SATD-C vs 42 SATD-I instances per repository, median values). The same behavior is observerd when analyzing relative distributions. In this case, SATD-C items are more frequently used as means to document SATD (i.e., 58% of SATD-C vs 41% of SATD-I per repository, median values). These observations are statistically confirmed by applying the Mann-Whitney U test (p-value ≤ 0.05).

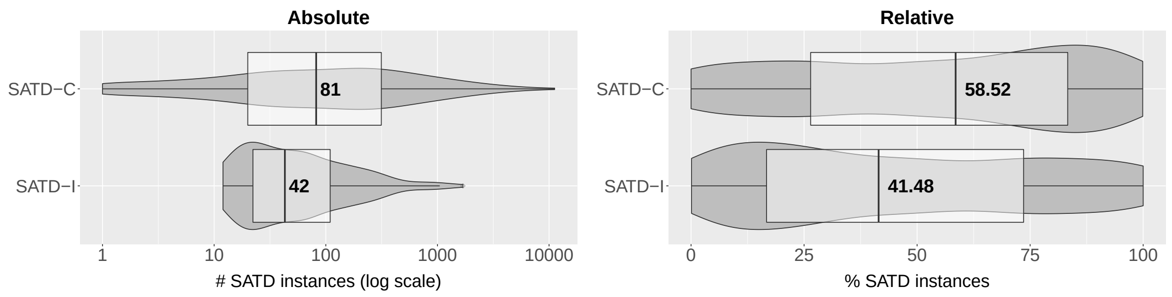


Figure 4.1: Distribution of SATD-I and SATD-C instances per repository, in terms of absolute and relative numbers.

To better characterize our dataset and shed light on key quantitative measures about both datasets of SATD instances, we defined four dimensions[7]:

- *Size:* represents the volume of text used to describe TD.
- *Age:* characterizes how long the debt remains in code.
- *Activity:* represents the level of activity to document TD.
- *Engagement:* characterizes community involvement.

SATD-C Characterization. In Figure 4.2, we provide violin plots to describe the SATD-C instances in our dataset. To measure the first dimension (size), we counted the number of characters used in comment texts. In this case, the first quartile, median, and third quartile are 30, 63 and 131 characters. For age, we calculated the difference (in days, rounded with two decimal places) between the date of our data collection and the commit date when the SATD-C was introduced. The introduction date was obtained after traversing back the git tree until we find the commit responsible for adding each debt. As a result, the obtained quartiles are: 448.24, 823.71, 1,465.41 days, respectively. To characterize the level of activity performed by developers to document SATD-C, we counted the

[7]However, we highlight that it is not possible to directly compare and contrast the two sets (SATD-C and SATD-I), since the metrics used in each one are distinct.

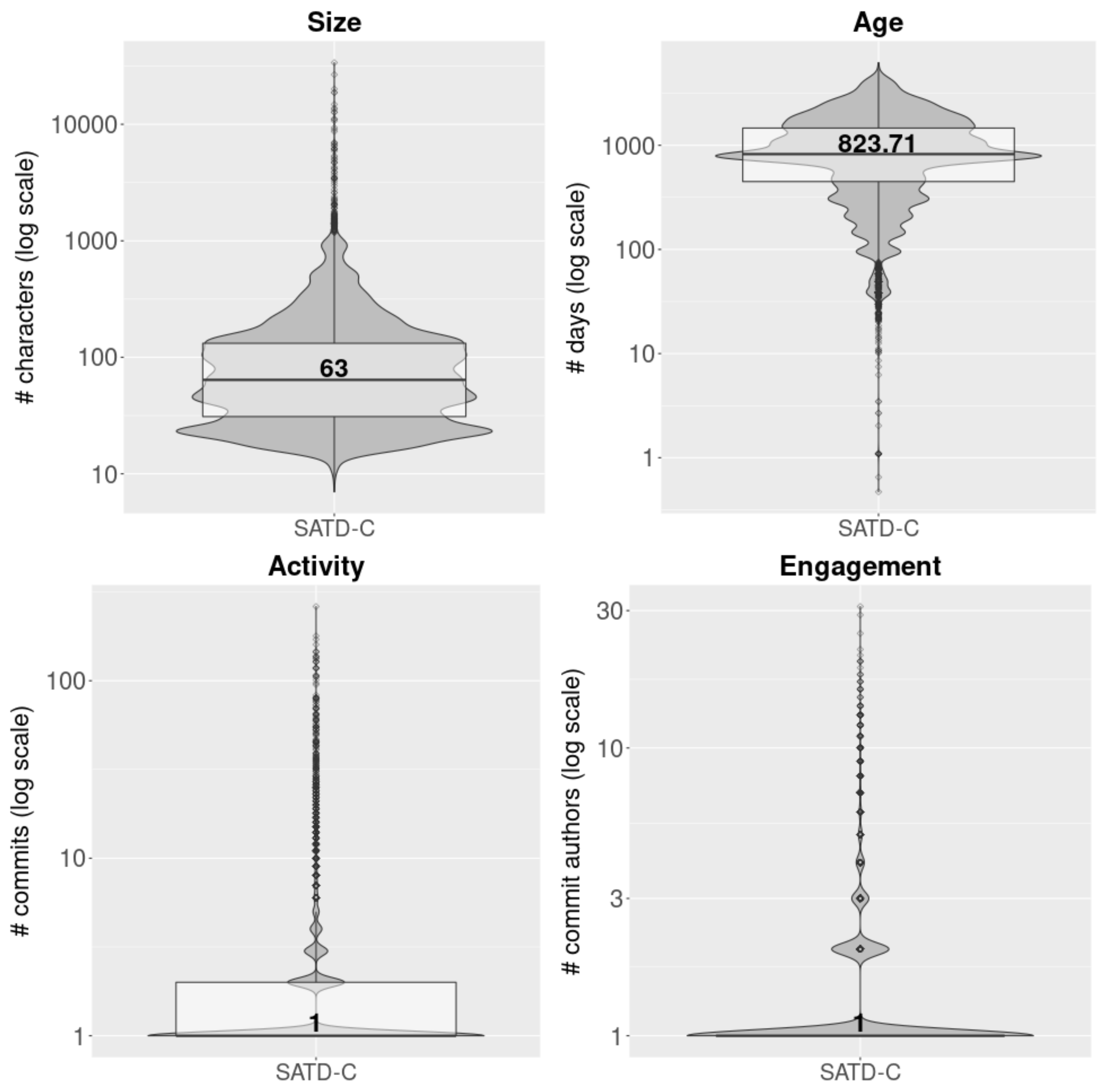


Figure 4.2: Characterization of Satd-C instances in terms of size, age, activity and engagement.

number of commits that modified the comment since its introduction. The first quartile, median, and third quartile are 1, 1 and 2 commits. Finally, to measure the community engagement, we counted the number of developers that modified the Satd-C comment. In this case, the three quartiles are equal to 1 author per Satd-C.

Satd-I Characterization. Figure 4.3 details the distribution of results for the metrics defined in each dimension, considering the Satd-I instances in our dataset. As for Satd-C, we used the number of characters as measure for the size dimension. The obtained quartiles are 203, 442 and 1,005 characters. To characterize the age of the studied Satd-I, we calculated the delta between the data collection and creation dates for the opened issues. In this case, the first quartile, median, and third quartile are 281.77, 671.22 and 1,251.18 days. Next, we selected the number of comments as a metric to characterize

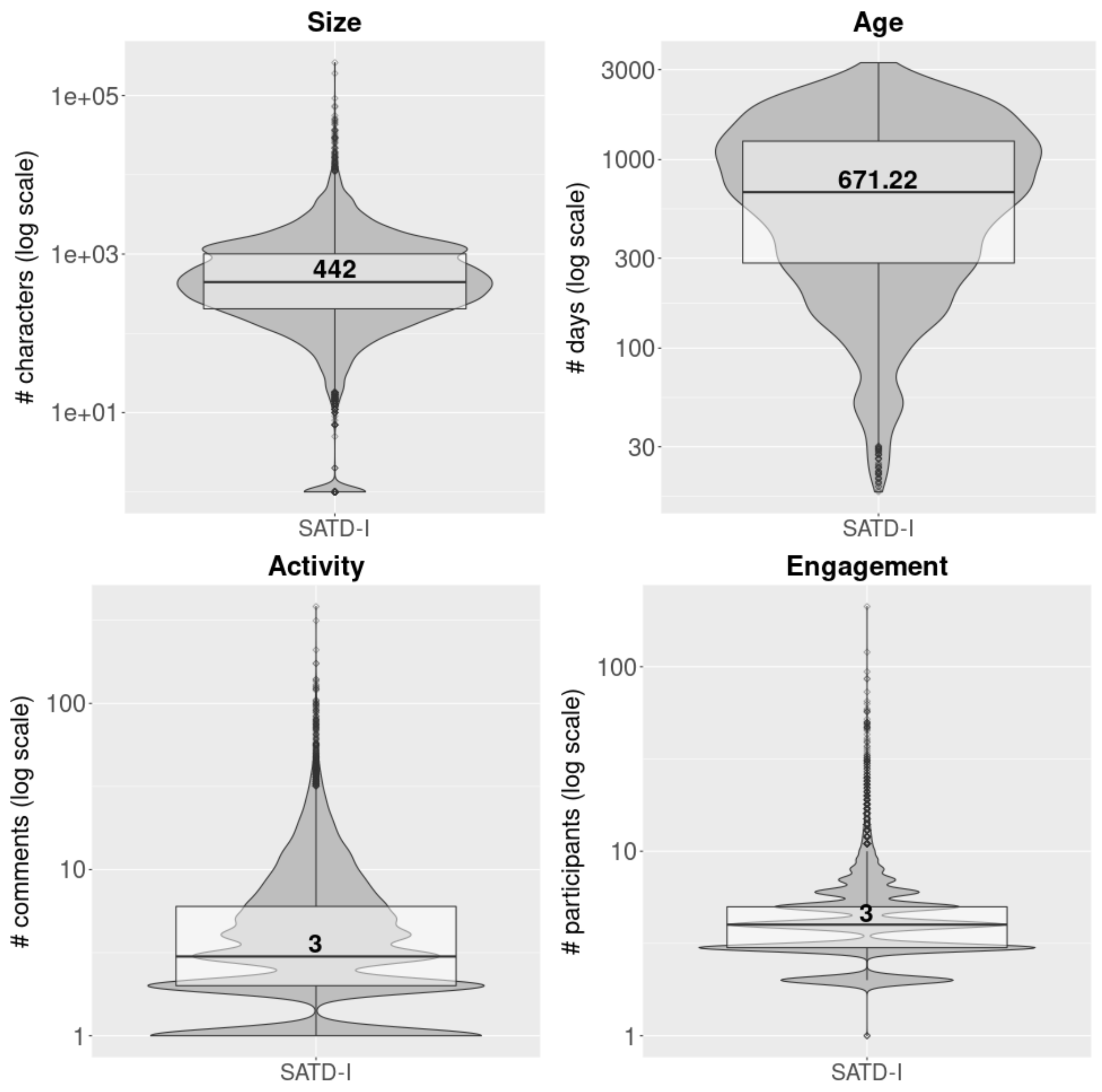


Figure 4.3: Characterization of SATD-I instances in terms of size, age, activity and engagement.

the activity in SATD-I issues. We observed 2 comments in the first quartile, 3 in the median, and 6 in the third quartile. Finally, we used the number of participants in the issue to describe the community engagement. The results for this dimension are 2, 3 and 4 participants per issue.

4.3 Transforming SATD-C in SATD-I

In this section, we assess developers' interest in tools to automatically create GitHub issues from Satd-C. Such tools might be effective to support the migration of Satd-C into Satd-I. In this case, developers can benefit from issue trackers features to manage Satd, such as: discussions, assignments, and increased visibility. Particularly, we aim to answer the following research question:

RQ4. Are developers interested in tools to create issues from Satd-C? To answer the RQ, we first implement and evaluate a prototype tool, called AdmiTD. This tool automatically identifies and reports Satd-C as GitHub issues. To evaluate its feasibility, we survey developers from 10 GitHub projects.

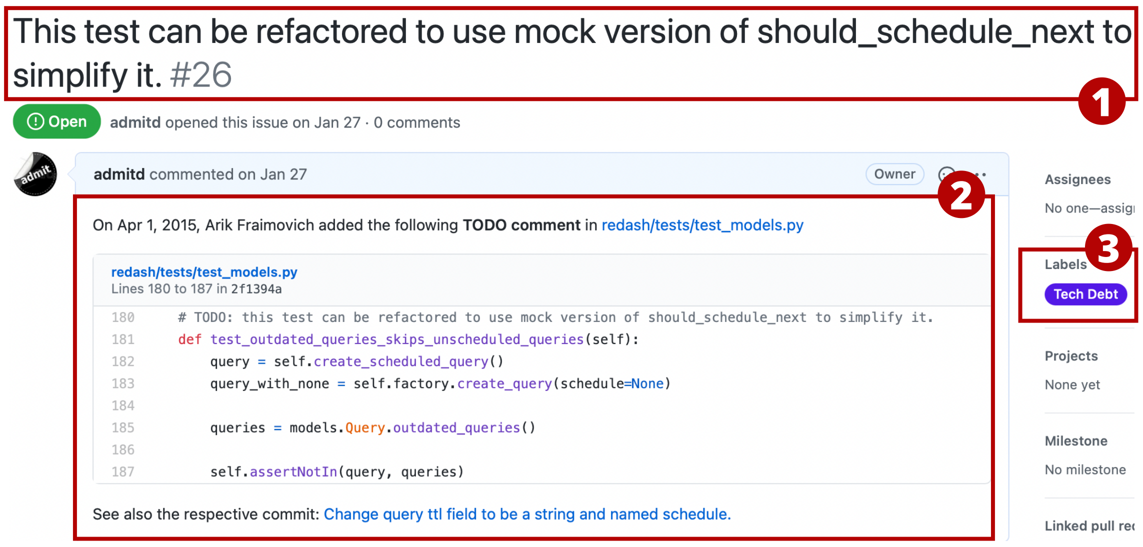
We dedicate Section 4.3.1 to present AdmiTD. Section 4.3.2 details our study design. Finally, in Section 4.3.3, we answer RQ4.

4.3.1 AdmiTD Tool

We first implemented AdmiTD, a prototype tool that automatically identifies and reports Satd-C as GitHub issues. To identify Satd comments, AdmiTD relies on the heuristics reported in Section 4.2.1 (i.e., we extract source code comments and search by common TD-related terms reported in the literature). For each identified Satd-C, our prototype tool automatically creates an issue in the analyzed repository, containing three major parts: (1) title; (2) body; and (3) label. To create the title of the issue (flagged with #1 in Figure 4.4), AdmiTD relies on the first sentence of the comment. Next, to create the body of the issue (flag #2), the tool analyzes the git log of the Satd-C lines and retrieves its introducing commit, author, and date. The issue body describes this information, as well as the code snippet of the debt. We fixed a length of at most the next 10 lines of code. Finally, the issue is automatically labelled with the TD-related label commonly used in the repository (flag #3).

Particularly, the issue shown in Figure 4.4 was created by AdmiTD for getredash/redash. It was generated from the following Satd-C introduced in April 1st, 2015 (see flags #1 and #2 in the figure):

```
# TODO: this test can be refactored to use mock version of
should_schedule_next to simplify it.
```


This test can be refactored to use mock version of should_schedule_next to simplify it. #26

admitd opened this issue on Jan 27 · 0 comments

admitd commented on Jan 27 Owner

On Apr 1, 2015, Arik Fraimovich added the following **TODO comment** in redash/tests/test_models.py

redash/tests/test_models.py
Lines 180 to 187 in 2f1394a

```
    # TODO: this test can be refactored to use mock version of should_schedule_next to simplify it.
    def test_outdated_queries_skips_unscheduled_queries(self):
        query = self.create_scheduled_query()
        query_with_none = self.factory.create_query(schedule=None)

        queries = models.Query.outdated_queries()

        self.assertNotIn(query, queries)
```

See also the respective commit: Change query ttl field to be a string and named schedule.

Assignees
No one—assig

Labels
Tech Debt

Projects
None yet

Milestone
No milestone

Linked pull re

Figure 4.4: Example of issue automatically created by AdmiTD, highlighting the generated (1) title, (2) body, and (3) label.

In this case, the Satd-C author reports that a referred test should be refactored to use mocks. Besides, the *Tech Debt* label was included as adopted in the repository (flag #3).

Finally, to improve the quality of the generated issues and reduce duplicates, AdmiTD merges issues related to the same code comment (i.e., Satd-C instances that would generate the same title). For that, issue bodies are appended with a separating line between each occurrence.

4.3.2 Survey Design

To explore developers interest in automatically transforming Satd-C in Satd-I, we conducted a survey with a sample of developers from 10 repositories selected in our dataset. For this, we first forked each repository, and applied AdmiTD in the fork (to avoid polluting the original repository with possibly unwanted issues). We also added two restrictions in AdmiTD to highlight relevant debts and facilitate developers' evaluation: (i) we only created issues for Satd-C instances retrieved from relevant files (i.e., files that concentrate 80% of the source code changes in the repository); and (ii) we limited the number of Satd-I to 25 issues, randomly selected (i.e., we restricted the report to the first page of GitHub issue tracker).

Table 4.1 shows the repositories selected to answer this second RQ, the number of existing Satd-C and Satd-I, as well as the number of issues generated by AdmiTD.

As we can see, the tool created 154 issues for 1,314 SATD-C instances. The difference between the number of generated issues and SATD-C instances is due to the restrictions for relevant files, and due to the maximum size of GitHub issue pages, as well as to the merge approach implemented by ADMITD. For example, in MKDOCS/MKDOCS two issues were merged and one SATD-C does not occur in a relevant file. All issues can be found at the forked repositories listed at ADMITD GitHub page.[8]

Table 4.1: SATD-I instances automatically generated by ADMITD.

Repository	**# SATD-C**	**# SATD-I**	**# Gen.**
OSQUERY/OSQUERY	45	22	25
BALDERDASHY/SAILS	53	216	25
GETREDASH/REDASH	834	40	25
FALCONRY/FALCON	51	39	25
NAVER/PINPOINT	280	48	23
GABIME/SPDLOG	13	19	11
ENCODE/DJANGO-REST-FRAMEWORK	14	63	6
GIONKUNZ/CHARTIST-JS	12	15	6
APPIUM/APPIUM	5	37	4
MKDOCS/MKDOCS	7	16	4
Total	**1,314**	**515**	**154**

After generating the aforementioned issues, we emailed the core developers of each repository, sharing the link of the issues in our forked repository, explaining our tool and goals, and asking a single question: *Is it worthwhile to create such issues in your repository?*

4.3.3 Are developers interested in tools to create issues from SATD-C?

From a total of 10 mails sent, we received four answers (response rate of 40%). However, none of the received answers included strong evidence of a positive perception of the automatically generated issues. In other words, although developers use GitHub issues to document and discuss their debts, they may not be interested in the automatic SATD-C→SATD-I transformation. For example, the core developer of MKDOCS/MKDOCS commented on the noise produced by SATD-I issues:

[8] `https://github.com/admitd`

This is an interesting idea, but I don't think it is appropriate for mkdocs. Some projects may use a workflow where they would like to track each TODO as a bug. However, generally speaking I think that it will just create noise. For me a TODO is something that can be improved or fixed, but it isn't urgent. So if a developer spots it and has time they can take a look. Adding the noise of Github doesn't seem useful.

He also commented on the usage of IDE-based tools to search and index SATD-C instances:

Many IDEs or other developer tools can let you easily view these. I think that is enough.

A relevant example of such tools is the TODO TREE: an extension for VS Code with more than 1M installs.[9]

In addition, the core developer of OSQUERY/OSQUERY discussed the easiness of annotating TD in code comments to speed up development:

I think having TODOs/debt annotated in code is OK and we want to encourage folks to ship (tested) code quickly without having to be perfect or with all features implemented.[10]

> Developers did not provide strong evidence of being interested in tools to automatically create SATD-I from SATD-C.

4.4 Linking SATD-C to SATD-I

Given the negative results in Section 4.3, we decided to study another approach to support developers handling both SATD-C and SATD-I. Instead of creating new SATD-I from SATD-C, we investigate the viability of implementing tool support for connecting SATD-C to existing instances of SATD-I. We claim that such tools can make the navigation from SATD-C to SATD-I easier and straightforward. Therefore, these tools might help developers to include more details about a given debt, and to keep track of duplicated and outdated reports.

Our first step is to assess how often SATD-I issues are referenced from SATD-C comments (SATD-C→SATD-I). Thus, we seek to answer the following research question:

[9]`https://marketplace.visualstudio.com/items?itemName=Gruntfuggly.todo-tree`

[10]Interestingly, this comment suggests that previous studies on SATD-C might have maximized the occurrence of TD, by considering any *TODO* to be problematic. As stated by the respondent, a *TODO* can just be a minor observation for the developer in the future.

RQ5. Do developers refer to SATD-I in SATD-C? To answer this last RQ, we search for explicit links between SATD-C and SATD-I. For this, we mine the occurrences of SATD-I's URLs and IDs in SATD-C's comments. The existence of such links would mean that it is feasible to create such tool.

We dedicate the remaining of this section to describe the methods applied to find these links, as well as the obtained results.

4.4.1 Do developers refer to SATD-I in SATD-C?

We implemented a custom procedure to triangulate the debts from both groups. Basically, we implemented a script to analyze the code comments of each SATD-C and to extract the numbers that matched one of the following conditions:

(a) Considered as a single token, such as *"TODO: Issue 949 - the following code . . . "*.

(b) Preceded by the `#` symbol, e.g., *"TODO #7967 help refactor"*.

(c) Preceded by the `/issue/` token, for example: *"TODO: change to 200 **https://github.com/loadimpact/k6/**issues/1250"*.

Then, we cross-check these numbers with the issue codes of the SATD-I collected in Section 4.2 to link both SATD-C and SATD-I. SATD-C instances referring to issues outside of the extended dataset were discarded as they do not fulfill the SATD-I criteria adopted initially, i.e., contain at least one TD-related label. Lastly, the author of this thesis manually inspected each link to ensure they indeed represent actual SATD instances (which indeed was confirmed in all cases).

From the 190 repositories initially analyzed, 23 of them contained at least one SATD-C→SATD-I occurrence (12.1%). Figure 4.5 presents the number of SATD-C and SATD-I collected in these repositories, as well as the number of SATD-C→SATD-I occurrences we found. Our triangulation process matched 80 references in total. From the perspective of SATD-C, this means that 0.36% of them refer to a SATD-I instance; conversely, 1.28% of SATD-I is referenced by at least one SATD-C instance in our dataset.

Table 4.2 lists the number and distribution of these references for each of the 23 repositories. COCKROACHDB/COCKROACH has the highest number of references (20), followed by RADAREORG/RADARE2 (11), and both ELASTIC/KIBANA and KUBERNETES/KUBERNETES, with 8 instances, each; together these four hold 58.8% of all SATD-C→SATD-I links. By contrast, 13 repositories have only one reference. Despite this high concentra-

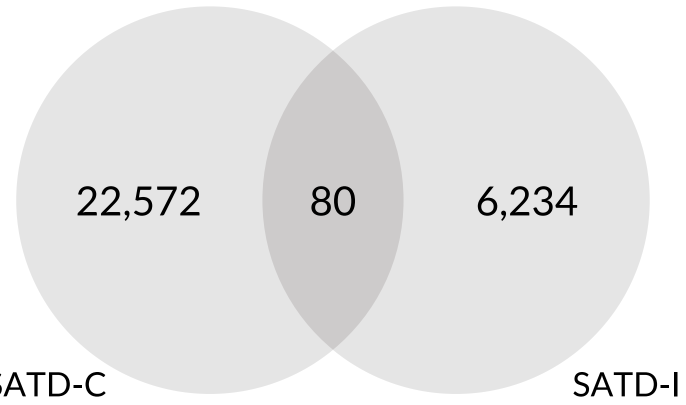


Figure 4.5: Number of Satd-I and Satd-C instances considering only repositories with at least one Satd-C→Satd-I reference.

tion, the occurrence of Satd-C→Satd-I represents less than 5% in the top-4 from both Satd-I and Satd-C perspectives.

Table 4.2: Number of Satd-C→Satd-I references for each repository.

Repository	**# Ref**	**% Satd-C**	**% Satd-I**
cockroachdb/cockroach	20	0.61%	2.79%
radareorg/radare2	11	0.62%	4.06%
elastic/kibana	8	0.70%	2.69%
kubernetes/kubernetes	8	0.21%	0.92%
adobe/brackets	5	0.78%	3.29%
loadimpact/k6	5	0.71%	7.25%
dotnet/roslyn	3	0.23%	1.20%
kubernetes/minikube	3	6.38%	1.97%
angular/angular-cli	2	0.41%	1.27%
withspectrum/spectrum	2	0.93%	1.57%
microsoft/vscode	1	0.23%	0.06%
grpc/grpc	1	0.32%	0.36%
metabase/metabase	1	0.28%	0.45%
hashicorp/consul	1	0.08%	1.30%
influxdata/influxdb	1	0.21%	0.71%
firecracker-microvm/firecracker	1	5.26%	0.83%
microsoft/react-native-windows	1	0.26%	1.25%
elastic/beats	1	0.34%	5.56%
openshift/origin	1	0.03%	0.34%
jetstack/cert-manager	1	0.35%	2.38%
WordPress/gutenberg	1	0.74%	0.71%
tektoncd/pipeline	1	0.10%	2.08%
perkeep/perkeep	1	0.13%	8.33%
Total	**80**	**0.36%**	**1.28%**

After manually inspecting the content of such SATD-C→SATD-I occurrences, we noted that SATD-C were generally created to highlight the points in the code impacted by the associated SATD-I instance. For instance, the following SATD-C was identified at KUBERNETES/KUBERNETES:

```
#TODO refactor all tests to use real watch mechanism, see #72327
```

The referenced SATD-I, in turn, describes the need to *"Use fake client real watch mechanism in PV controller tests"*.

In any case, mentioning SATD-I in SATD-C is not a widespread practice in GitHub projects. Even after reducing our analysis to the few projects containing such a link, we observe that such referral is, in fact, barely used. Moreover, an in-depth analysis showed that 26 references (out of the 80 references initially detected) refer to duplicate links. In other words, 26 SATD-C comments refer to duplicate issues. This result suggests that SATD-I may be used to document TD that spans in multiple places in code.

> Overall, only 80 out of 22,327 (0.36%) SATD-C explicitly refer to SATD-I. Thus, cross-referring SATD-C and SATD-I is not a widespread practice.

4.4.2 Other Matching Approaches

In order to try to expand our dataset of SATD-C→SATD-I references, we analyzed more flexible matching techniques. The idea was to search for other traces that could relate both SATD. In this sense, we decided to try out two distinct approaches: *textual similarity*, where a pair of SATD-C→SATD-I is created in case they are textually similar to each other; and *timestamp proximity*, where the link is generated if both SATD are created next to each other given a time interval.

Textual Similarity. For each pair of SATD-C and SATD-I, we assume that the former is semantically related to the latter whenever there is a high textual similarity between them. For this, we relied on the *SequenceMatcher* class, available at the *difflib*[11] Python module; this class provides functions to compute the textual similarity level between two string sequences, returning a score in $[0, 1]$ range. We calculated the ratio for all possible SATD-C→SATD-I combinations, considering the texts retrieved from SATD-C comment blocks and SATD-I issue bodies. Then, we filtered the pairs with a ratio of 0.75 or higher. However, *we did not find any references based on these criteria.*

[11]https://docs.python.org/3/library/difflib.html#module-difflib

Timestamp Proximity. We considered that one SATD-C instance refers to a SATD-I if the latter was created at most 24 hours after the creation of the former; as a result, we leveraged a total of 879 pairs through this pattern. To verify the effectiveness of this approach, we selected a random sample of 87 occurrences and analyzed each one in order to validate this temporal connection.[12] However, *we were not able to identify any real association.* To illustrate this finding, consider the following SATD-C:

```
#Workaround for https://github.com/microsoft/vscode/issues/12865
check new scrollY and reset if necessary
```

As noted, this comment clearly mentioned issue #12865, but it was wrongly linked with issue #15515, as they were created in an interval of three hours.

Mixed Approach. We performed one last experiment by combining both approaches. This time, we considered a SATD-C→SATD-I as valid if they were created in a 24-hour window, but have a similarity of 0.30 or higher. As a result, this procedure returned 12 references. The author of this thesis manually checked each one, but *observed that they were all invalid.*

> More flexible approaches did not improve our search for SATD-C→SATD-I references. We were not able to find any new reference using such approaches.

4.5 Implications

Based on our results, we shed light on the following practical implications:

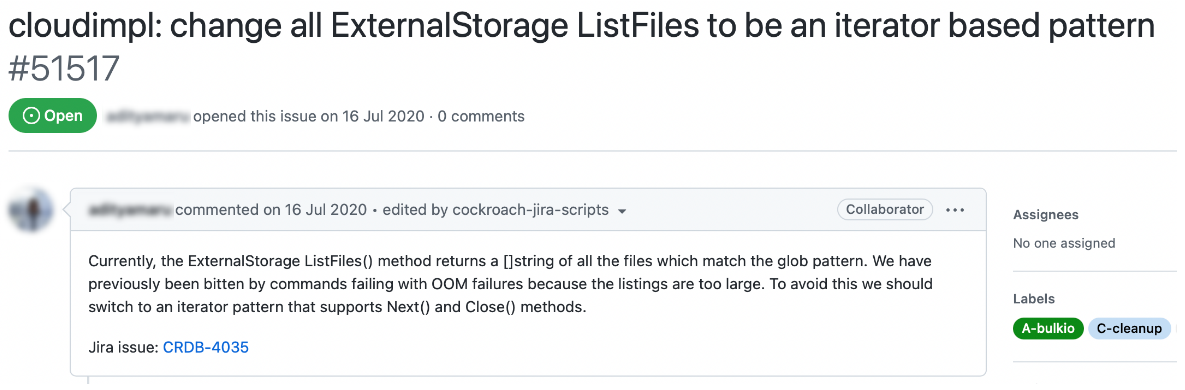
1. SATD-C and SATD-I probably have different natures. This difference could explain the negative results we reported when investigating tool support for SATD-I (RQ4 and RQ5). We hypothesize this result happened due to the distinct natures of SATD-C and SATD-I. In this case, we believe that SATD-C could be more adopted to report low-level TD (i.e., debts associated with code snippets located next to the TD comment). For example, in the following SATD-C comment, a developer from ELASTIC/KIBANA indicates that a particular excerpt of the code is duplicated and should be cleaned up:

```
// TODO: everything below performs verification of manifest.yml
// files, and hence duplicates functionality already implemented
// in the package registry.  At some point this should probably
```

[12]Determined after specifying a limit of 95% confidence level, with a margin of error of 10%.

```
// be replaced (or enhanced) with verification based on
// https://github.com/elastic/package-spec/
```

On the other hand, SATD-I could be more suitable to document high-level concerns (i.e., debts that tend to spread out over the code, such as the ones referring to design concerns). Figure 4.6 illustrates an example of a high-level TD concern documented through an issue. In this case, the developer from COCKROACHDB/COCKROACH created this SATD-I to report a debt that spans in multiple places of the code.



cloudimpl: change all ExternalStorage ListFiles to be an iterator based pattern #51517

Open [redacted] opened this issue on 16 Jul 2020 · 0 comments

[redacted] commented on 16 Jul 2020 • edited by cockroach-jira-scripts

Collaborator

Currently, the ExternalStorage ListFiles() method returns a []string of all the files which match the glob pattern. We have previously been bitten by commands failing with OOM failures because the listings are too large. To avoid this we should switch to an iterator pattern that supports Next() and Close() methods.

Jira issue: CRDB-4035

Assignees

No one assigned

Labels

A-bulkio C-cleanup

Figure 4.6: Example of high-level SATD in COCKROACHDB/COCKROACH issues.

Moreover, we also hypothesize that the decision for adopting code comments or issues may also depend on other factors, such as the priority, impact or importance of TD problems. Therefore, we highlight that it would be relevant to empirically validate these hypothesis, supporting developers to better document TD.

2. Issues might be more useful to report TD related to crosscutting concerns. In addition to the previous implication, we conjecture that issues might be more useful to document crosscutting TD. Particularly, in RQ1 we show that 40% of the studied SATD-I instances were related to TD more challenging to be documented in code comments (e.g., UI, PERFORMANCE, and BUILD debts). In fact, this result complements previous SATD-C studies—that mainly identified SATD types related to source code—and prospects new horizons to SATD research, mainly related to less studied TD types.

3. Tools for linking SATD types might not be worthy. Still based on our negative results in RQ4 and RQ5, the implications to tool builders seem to be clear: they should not invest in tools to connect both types of SATD, using approaches similar to ADMITD. In terms of SATD-C tools, the most promising ones are related to indexing SATD-C instances, as mentioned by one of the surveyed developers. Similarly, we also envision similar tools for indexing SATD-I instances.

4. Researchers should include SATD-I in SATD studies. Although there is a relevant difference between the amount of TD documented in GitHub issues and in code comments (42 SATD-I and 81 SATD-C instances per repository, on the median), we consider that issues must be considered by Satd researches, as they tend to represent Technical Debt in a different shape. Therefore, it is recommended that studies on Satd rely not only on debts reported in comments (as usual), but also include instances documented in issue trackers.

4.6 Threats to Validity

First, our new dataset is based on a list of 219 TD-related labels, manually elicited from labels present in the top-5K most popular GitHub repositories. Although this list is based on the state-of-the-art [Xavier et al., 2020], we acknowledge that it is not exhaustive. Moreover, we relied on a conservative set of terms to identify Technical Debt in code comments, such as: *TODO*, *workaround*, *fixme*, and *hack*. Therefore, it is possible that other Satd-C instances were not selected in our study, i.e., comments with different keywords. Even though, we claim that our final dataset is one of the largest in the literature, including more than 20K Satd-I and 72K Satd-C instances.

Another threat that would affect this study refers to the decision to rely on developers' self-admission to identify both Satd-I and Satd-C. We mitigate it by adopting a conservative list of TD-related keywords to mine Satd-C, and by manually analyzing the GitHub labels used to document Satd-I. Additionally, the author of this thesis manually inspected a relevant subset of the final dataset. We also mitigate the bias of developers' answers by restricting our survey to core developers.

We also acknowledge that it is possible that some Satd-I instances in our extended dataset do not refer to TD concerns. For example, developers' misunderstanding of the definition of the TD concept may result in incorrectly labeled issues. Our strategy to identify Satd-C comments may also represent a threat, since we adopted a simple and straightforward pattern-based approach. For both threats, we claim that we relied on conservative decisions to mitigate the occurrence of false-positives (i.e., we selected a precise set of TD-related labels, as well as a meaningful set of TD keywords).

Additionally, we highlight that this study is limited to the 20,265 issues and 72,669 comments classified as Satd. Despite the size of the studied dataset and the relevance of their respective repositories (10K stars, on the median), our findings may not be generalized. Moreover, the results discussed in RQ6 are based on the impressions of few developers. However, we claim that the received answers provided valuable insights.

Regarding our study to answer RQ7, we acknowledge that our strategies to match Satd-C and Satd-I are not exhaustive. In fact, developers can adopt different approaches to link them (e.g., by using a tertiary software artifact). Moreover, to assess textual similarity, we deal with the construct validity by adopting well-known functions in the Python community. Finally, we highlight that we set relaxed thresholds for the textual similarity ratio (0.75 and 0.30) in order to increase the chances of finding references. Even with such decisions, we could not improve our search.

4.7 Final Remarks

In this chapter, we explored the interplay between SATD-C and SATD-I instances by investigating the adoption of tools to report TD. For that, we first built a dataset of 20,265 SATD-I instances and 72,669 SATD-C instances, mined from 190 well-known GitHub projects (e.g., VUEJS/VUE, MICROSOFT/VSCODE and KUBERNETES / KUBERNETES). We also implemented ADMITD, a prototype tool that automatically identifies and reports SATD-C in GitHub issues. We used this dataset and tool to (1) explore developers interest in automatically creating GitHub issues based on SATD-C; (2) investigate whether developers cross-reference SATD-I in SATD-C. As a result, we show that:

- Developers are not interested in tools to automatically create SATD-I from SATD-C. We attempted to use ADMITD in 10 GitHub repositories without success;
- It might not be feasible to create a tool to link SATD-C and SATD-I instances. When looking for such references in our dataset we find out that this is not a widespread practice, i.e., it happens for less than 1% of the SATD-Cs.

Replication Package. We provide the complete dataset used in this chapter and a replication package at: `https://doi.org/10.5281/zenodo.6532378`.

Chapter 5

Documentation Guidelines

In this chapter, we investigate the circumstances that drive developers to choose between code comments and issues when documenting Technical Debt. Particularly, we conduct a survey with authors of SATD-C and SATD-I instances to elicit a catalog of guidelines to better document SATD. In Section 5.1 we introduce the chapter. In Section 5.2 we detail the survey design. Section 5.3 presents the obtained catalog. Finally, in Sections 5.4 and 5.5 we discuss threats to validity and conclude the chapter, respectively.

5.1 Introduction

In Chapter 4, we showed that there is a minor interplay between SATD-C and SATD-I. Particularly, our results indicate that code comments and issues are used by developers to document Technical Debt in distinct scenarios. However, **there is still a lack of knowledge on the circumstances that drive developers to choose between them**. In this chapter, we tackle this problem by surveying developers who documented TD using both comments and issues, as they have practical experience with both forms of SATD.

To accomplish that, we first leverage the authors of both SATD-C and SATD-I instances from our large-scale dataset (Section 4.2). From a set of 1,006 developers, we contacted 137 developers to answer the following research question:

RQ6. Where do developers document TD? In this RQ, we conduct a survey with developers that documented TD using both comments and issues. We received answers from 59 authors who created both SATD-C and SATD-I instances in our dataset. Specifically, we aim to unveil guidelines to support developers to better document TD.

5.2 Study Design

Our ultimate goal is to conduct a survey to reveal how developers select between issues and comments to report Technical Debt. For that, we first we leveraged the list of developers who created at least one of the Satd instances collected in our large-scale dataset (Section 4.2): the author who opened the issue for Satd-I, and the commit author for Satd-C. We identified 8,082 authors who reported these debts in the 20,265 issues and 72,669 comments available in our extended dataset. We used the author name, as provided in GitHub profile and in git commit, to identify authorship and then compute the intersection of authors who adopted both strategies. Figure 5.1 depicts the relationship between the authors of each group, considering the whole dataset. As we can observe, 3,243 (40%) authors reported Satd-C only, while 3,833 (47%) authors reported only Satd-I. Finally, 1,006 authors (12%) reported both.

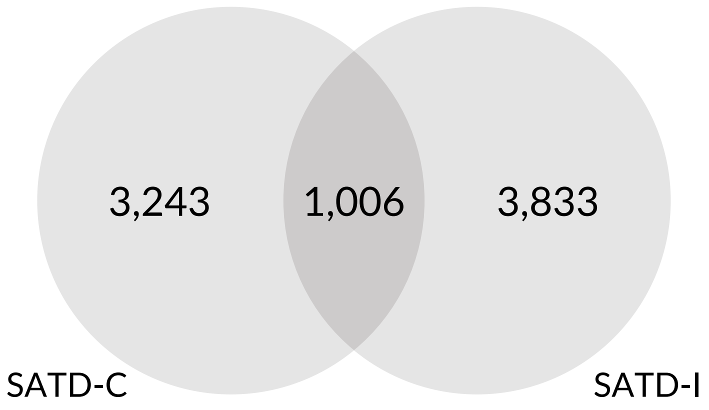


Figure 5.1: Number of developers who reported each type of Satd.

As we are primarily interested in comparing the motivations for choosing between Satd strategies, we considered only authors who created both Satd-I and Satd-C, i.e., 1,006 authors in the intersection. From this total, we selected the ones who (i) created both Satd instances in the previous 1.5 years (from September 3rd, 2019 to March 3rd, 2021); and (ii) provided their email address publicly in GitHub. Overall, we contacted 137 developers from 51 repositories.

In each email, we first presented both Satd-C and Satd-I instances authored by the developer (as a GitHub permanent link). In situations where developers created more than one instance of Satd in the studied time frame, we used the most recent one. Next, we asked two open-ended questions:

1. *When do you recommend documenting TD using code comments?*
2. *When do you recommend opening an issue?*

We received 52 answers, which represents a response rate of 38% (52 answers to 137 inquiries). Furthermore, after being reached by our email, two participants considered our research interesting and asked to share the discussion in the Slack channel of their repositories. We then included seven answers received from this "snowballing phase", resulting in a total of 59 answers.

Finally, the author of this thesis followed an open-card sorting [Spencer, 2009] approach to extract guidelines from the survey answers. We decided to follow this method because it allows the emergence of themes based on the qualitative analysis of answers. It consists in the following steps: (i) identifying themes from answers, (ii) reviewing the themes to find opportunities for merging, and (iii) defining and naming the final themes. Specifically, the author of this thesis first analyzed each answer and extracted 18 themes. Next, these themes were reviewed and merged into 13 semantically equivalent themes. In a final step, they were packed into the guidelines presented in Section 5.3. For example, in the first round, the themes ACKNOWLEDGES TD IN CODE REVIEW and ADDS HINTS TO THE READER were elicited. In the second phase, they were merged and, finally, rephrased as IF IT PROVIDES CONTEXT TO THE READER. To conclude this analysis, a research collaborator independently analyzed the 59 answers. He agreed with all the proposed guidelines. However, in ten cases, he argued the answers also discuss additional guidelines, which were then included in our final classification.

5.3 Where do developers document TD?

As presented in Table 5.1, we identified 13 guidelines, divided in two categories: six guidelines to document TD using comments (SATD-C); and seven to create issues (SATD-I). To better discuss each guideline, we labeled them with a unique identifier (C#ID for SATD-C and I#ID for SATD-I).

In most cases, a given answer produced more than one guideline. This explains why the total number of occurrences is higher than the number of answers (59 answers). Next, we detail the catalog explaining each guideline and illustrating them with quotes from developers. We label quotes with D1 to D59 to indicate developers answers.

Table 5.1: Guidelines to document SATD.

It is recommended to use...		Occ.
SATD-C	C1. IF IT PROVIDES CONTEXT TO THE READER	34
	C2. IF IT HAS LOW PRIORITY	14
	C3. IF IT HAS A LOCAL SCOPE	11
	C4. IF IT REQUIRES SMALL EFFORT TO FIX	8
	C5. IF IT WILL BE ADDRESSED SOON	5
	C6. IF YOU REVISIT THE CODE FREQUENTLY	2
SATD-I	I1. IF IT REQUIRES DISCUSSION	18
	I2. IF IT NEEDS TO BE TRACKED	16
	I3. IF IT SPANS TO MULTIPLE PLACES	15
	I4. IF IT REQUIRES VISIBILITY	15
	I5. IF IT HAS HIGH PRIORITY	10
	I6. IF IT REQUIRES MEDIUM/LARGE EFFORT TO FIX	8
	I7. IF IT IS A GOOD FIRST ISSUE	5

5.3.1 Guidelines for Using Comments

We elicited six main guidelines for using comments as means to document TD. In general, developers suggest that it is preferable to rely on source code comments to provide additional context to code debts, and to document low priority or local concerns. Specifically, developers suggest that it is recommended to use comments:

C1. IF IT PROVIDES CONTEXT TO THE READER. With 34 answers (58%), the most discussed advice for SATD-C is related to including details about implementation decisions. In this case, authors should provide hints that allow future readers to understand workarounds or refactor the code to better solutions, as follows:

TODOs can also be helpful to explain a hacky implementation so that a future reader of the code can improve it or at least understand why the original implementer made the choice that they did. (D9)

I would recommend documenting TD using code comments when the information helps to understand the code by giving additional context for either myself or a colleague in the future, but is only necessary information in this very local context. (D26)

C2. IF IT HAS LOW PRIORITY. In 14 answers (24%), developers suggest that it is preferable to use comments for low priority TD. In this case, they claim that paying these debts happens by chance when other developers pass through the comment. D10 and D25 illustrate this guideline:

I left this as a TODO comment because it was a small implementation detail and I didn't see it as an important issue to tackle. (D10)

TODOs should be for short, one-off examples of tech debt that aren't a high priority to tackle rightaway. (D25)

C3. If it has a local scope. For 11 developers (19%), comments should be used to document local and specific debts. For example, D2 cite this guideline:

I would use a FIXME-like comment for something local to the code where the comment is (like a rare edge case not handled which should be handled near the comment). (D2)

C4. If it requires small effort to fix. Eight developers (14%) recommend to use Satd-C to document debts that would not require a significant effort to pay. For example:

If it's something fairly small (which will take <1h), but that you don't want or can't spend time doing at that moment. (D17)

C5. If it will be addressed soon. In five answers (8%), developers recommend to use code comments to document debts that will be removed in a short time. D47 illustrates this guideline:

When you're writing a lot of temporary code that you know will change in a few days, so it is pointless to open issues just to close them tomorrow. (D47)

C6. If you revisit the code frequently. Two developers (3%) highlight that comments should be used when they are constantly in touch with the debt. For example:

I would use comments in small projects where I have control over the whole code and I revisit the code frequently. (D6)

5.3.2 Guidelines for Using Issues

Our catalog also includes seven guidelines to document TD in issues. Generally, our analysis shows that developers use Satd-I to document debts that needs to be better discussed with other contributors or tracked by managers. Developers suggest to report debts as issues in the following scenarios:

I1. If it requires discussion. The most discussed recommendations for Satd-I (18 answers, 31%) refers to using issues to gather discussions with other contributors. Particularly, developers highlight that issues are preferable to document debts that need to be discussed to find better solutions, make clarifications or explore management alternatives (e.g., their priority). For instance:

It's also a way for other members of the project to put in their advice about the issue being discussed. (D23)

It enables discussions of technical debt in more abstract terms, as the documentation is not tied to the code. This allows other developers to provide their input and thus collaboratively allow a team to find solutions. (D31)

Writing issues makes it easier to collaborate on solutions, make clarifications, and gather information before doing the work. (D59)

I2. If it needs to be tracked. In 16 answers (27%), developers argue that issues are useful to support TD management as they are better tracked than code comments. For example:

Opening tech debt issues also helps to get data about the code quality of a project that can be used to convince management to invest in either cleaning-up time, or a refactor/rewrite of the code. (D26)

It allows us to measure at a project management level how much technical debt we've taken on (e.g. this week, we opened 5 technical debt issues, maybe we need to slow down development). (D39)

I3. If it spans to multiple places. Fifteen developers (25%) recommend that issues should be used to document debts that either occur in more than one location of the code or relate to abstract decisions. In both cases, finding one specific point in code to highlight the debt is not possible. This is illustrated as follows:

Fixing the TD would span multiple files and involve touching quite a lot of places in the codebase. (D2)

If TODO is more of architectural thing, spans multiple modules, and needs input from different teams, then I'd create an issue. (D5)

I'd say it's less about a specific hacky code block, and more about some more abstract design decision. (D23)

Anything larger that affects multiple parts of the code base should be an issue. (D25)

I4. If it requires visibility. For 15 developers (25%), issues should be used as a means to provide visibility to TD, preventing it from being forgotten in code. Developers D11 and D23 highlight this recommendation in the following answers:

An issue in the backlog is the actual 'should do this thing' record, that could cause it to actually get done. (D11)

Opening an issue is always better in the case of a community project, where the issue is far more visible/searchable than a code comment. (D23)

I5. If it has high priority. Ten developers (17%) contrast the usage of issues and comments according to their priority. In opposition to guideline C2, they argue that SATD-I should be used for high priority debts that ought to be paid. For instance, D30 illustrates this recommendation as follows:

I would say that issues are probably the most important form of communication for actually fixing the problem. So there should be an issue created for any tech debt which absolutely needs to be fixed. (D30)

I6. If it requires medium/large effort to fix. The cost for paying TD was also mentioned as criteria to decide for creating issues according to eight developers (14%). For example, D2 states:

An issue or tracker is for some bigger beast, like 'refactor this class' or 'change the way those classes interact'. (D2)

I7. If it is a good first issue. Finally, five developers (8%) included in their answers the recommendation of using SATD-I as a means to engage new contributors. D10 illustrates this recommendation in the following:

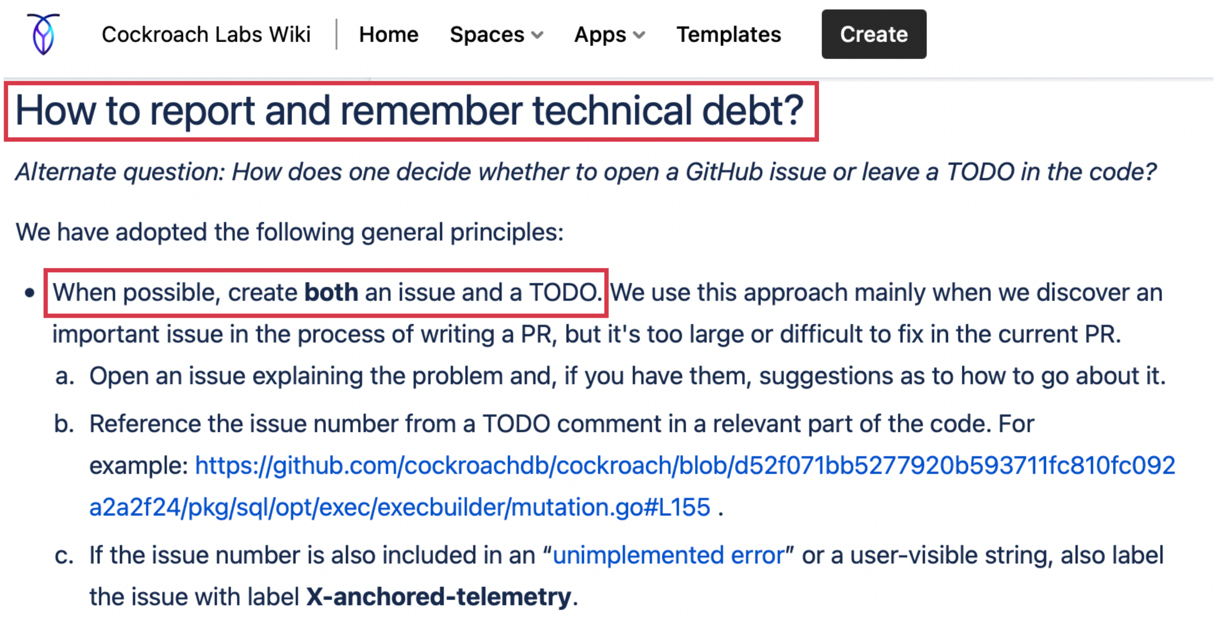
GitHub issues are particularly useful in the [project] because they can be tagged with '/good-first-issue' which enables new contributors to find things to work on and get familiar with the project. (D10)

5.3.3 Guidelines for Using Both Strategies

In 14 answers (19%), developers mention that the best practice to document TD is to mix both strategies. They argue that it is preferable to report TD in issues and make reference to them in comments. The goal of this mixed approach is to benefit from local documentation in comments and from features like discussion, tracking, and visibility in issues. D4 illustrates this guideline as follows:

There should be a 1:N relationship between issues and todos in the codebase. When looking at an issue, there needs to be a way to refer to all the locations in the code where a TODO *references that issue. The reverse direction also needs to be possible, i.e. when looking at a TODO, it should be easy to navigate to the issue tracking it.* (D4)

In fact, this mixed strategy was documented in COCKROACHDB/COCKROACH wiki after developers raised internal discussions on this topic due to our survey (in this repository, our questions were shared with contributors in their Slack channel, as we mentioned in Section 5.2). Figure 5.2 illustrates an excerpt of their recommendation. The wiki entry begins by proposing the best practice of adopting both strategies, but also highlights situations in which a single strategy is acceptable (guidelines C1, C2, I3, I6 in our catalog).



How to report and remember technical debt?

Alternate question: How does one decide whether to open a GitHub issue or leave a TODO in the code?

We have adopted the following general principles:

- When possible, create **both** an issue and a TODO. We use this approach mainly when we discover an important issue in the process of writing a PR, but it's too large or difficult to fix in the current PR.
  a. Open an issue explaining the problem and, if you have them, suggestions as to how to go about it.
  b. Reference the issue number from a TODO comment in a relevant part of the code. For example: https://github.com/cockroachdb/cockroach/blob/d52f071bb5277920b593711fc810fc092a2a2f24/pkg/sql/opt/exec/execbuilder/mutation.go#L155 .
  c. If the issue number is also included in an "unimplemented error" or a user-visible string, also label the issue with label **X-anchored-telemetry**.

Figure 5.2: Guidelines included in COCKROACHDB/COCKROACH wiki, after the discussion raised by our survey.

5.4 Threats to Validity

In addition to the threats reported in Section 4.6—related to the selection of SATD-C and SATD-I instances in our large-scale dataset as well as to generalization concerns—we highlight that the author of this thesis performed an open-card sorting to leverage the guidelines provided in RQ6. Despite the rigor followed by him to perform this classification, its replication may lead to different results. We mitigate this threat by constantly discussing the leveraged categories. We also mitigate the risk of bias in developers' answers by restricting our survey to developers who documented TD in the last year. These decisions were also important to increase the obtained response rates. Finally, the catalog provided in Section 5.3 is based on the impressions of 59 developers. However, we claim that the received answers provided valuable insights and deeply discussed the proposed questions, e.g., the biggest answer had more than 2.4K characters.

5.5 Final Remarks

In this thesis, we studied the circumstances that drive developers to document Technical Debt using code comments (SATD-C) or issues (SATD-I). To accomplish that, we surveyed 59 developers who authored both types of debts in a large-scale dataset containing 20,265 SATD-I and 72,669 SATD-C instances. We used the obtained answers to unveil practical guidelines to support developers to better document TD.

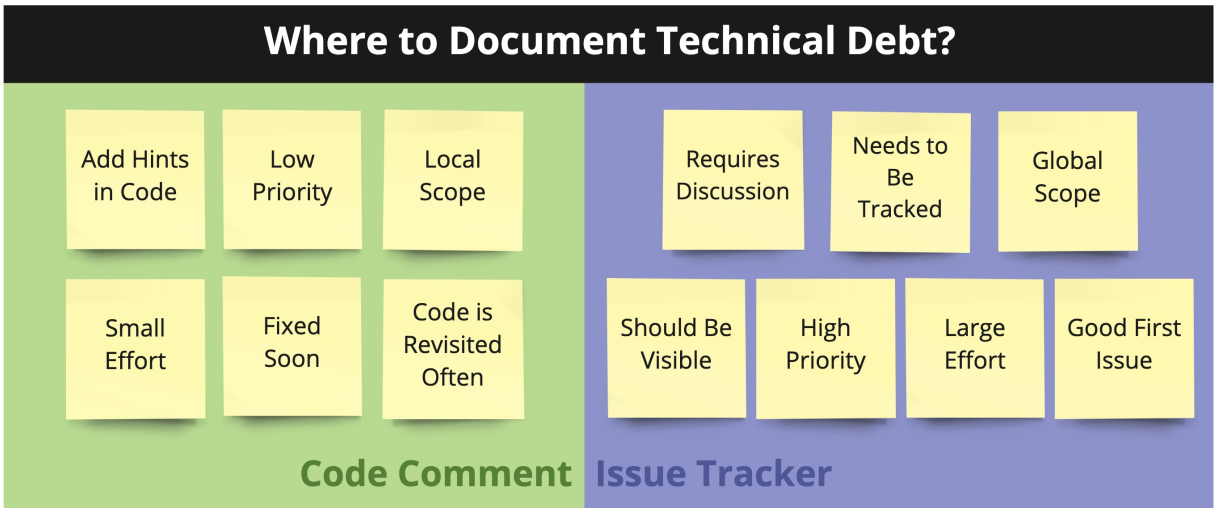


Figure 5.3: Technical Debt documentation guidelines.

The results of this work can directly benefit practitioners, since the leveraged guidelines provide empirical reference for choosing between issues, comments, or both when documenting TD. In fact, we summarized these guidelines in a cheat sheet, presented in Figure 5.3 and also available at `https://bit.ly/3HVZwVY`. Moreover, one project (COCKROACHDB/ COCKROACH) is already providing similar guidelines, upon being contacted by ourselves (see Figure 5.2). For researchers, our work shows that most developers (81%) report TD either as comments or issues, which reinforces the need to consider both situations when conducting empirical software engineering studies. Furthermore, our guidelines can help researchers when investigating solutions and tools to automatically detect debts in code and issues. Finally, educators may rely on this study to convey a list of best practices on how to report TD.

Before concluding, it is important to mention that in some projects issues are used by managers to take key development decisions. In this case, issues can be used for more critical TD that are of interest to managers; and comments can be used for low-level discussions that do not require managers' revision. By contrast, in very small projects, where the developers know and revisit the code frequently, using only SATD-C can be the most recommended practice. Finally, we also acknowledge that fixing TD, even the most simple instances, might be risky and have negative effects in other parts of the system.

Replication Package. The data used in this chapter is publicly available at `https://doi.org/10.5281/zenodo.6418088`.

Chapter 6

LTD: Less Technical Debt

In this chapter, we propose and assess LTD: a lightweight framework to support development teams to manage Technical Debt through the creation of issues. First, we introduce the chapter in Section 6.1. Next, in Section 6.2 we better present the LTD framework and its activities. Next, in Section 6.3, we detail the case study conducted to assess its adoption in a real-world scenario. In Section 6.4 we present the results of the study. Sections 6.5 and 6.6 discuss implications and threats to validity, respectively. Finally, Section 6.7 concludes the chapter.

6.1 Introduction

In this thesis, we aim to study the adoption of issue tracker systems to document and manage Self-Admitted Technical Debt. To accomplish that, we first explored SATD documentation in issues. Therefore, in Chapter 3 we characterized this practice by analyzing an initial dataset of 286 issues marked with a TD-related label. In this case, we investigated the types of SATD documented in these artifacts, as well as the motivations behind their introduction and payment. In Chapter 4, we advanced this understanding by investigating the interplay between SATD items documented in code comments and issues. At this point, we concluded that both SATD-C and SATD-I seem to be used to document Technical Debt from different natures (e.g., with distinct characteristics and purposes). Finally, in Chapter 5 we elicited the circumstances that drive developers to choose between code comments and issues when documenting SATD.

As a final step to achieve our general objective, in this chapter we propose and assess LTD: Less Technical Debt framework. Particularly, our motivation is to encompass the knowledge produced in the previous studies into a lightweight framework that may support development teams to better document and pay Technical Debt. The general idea is to introduce TD-related concerns into well-adopted methodologies (e.g., Scrum, Kanban, XP, etc.) [Valente, 2020], in a non-invasive and customizable way. Therefore, the framework proposes to inject four activities into agile-based methodologies already adopted by a team: TD Consensus, TD Discovery, TD Planning, and TD Payment.

Figure 6.1 illustrates the main activities proposed by LTD. As we can observe, the first activity (flag #1 in the figure), TD Consensus, seeks to produce a common understanding among developers about TD concepts (i.e., TD definitions, types, and priorities for the team). During TD Discovery (flag #2), we propose that developers prospect the debts already existing in the system, documenting and prioritizing the identified items. For that, **LTD indicates that the team should follow the practice described in the previous chapters and document the debts in labeled issues (which now we call as TD Stories)**. The third activity (flag #3), TD Planning, aims to let developers select TD Stories to be paid. Finally, during TD Payment days (flag #4), developers should concentrate efforts to pay the debts selected during TD Planning.

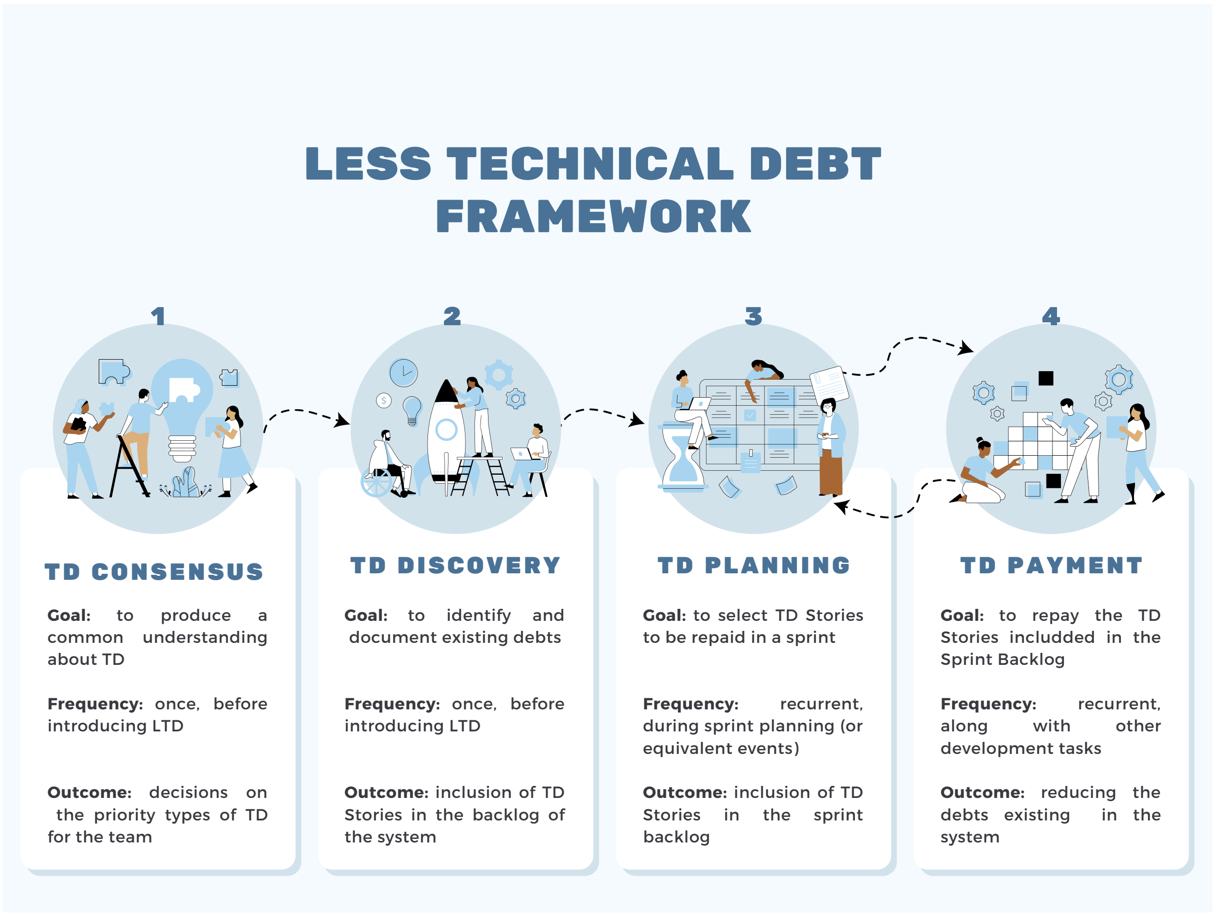

LESS TECHNICAL DEBT FRAMEWORK
1
2
3
4
TD CONSENSUS
Goal: to produce a common understanding about TD
Frequency: once, before introducing LTD
Outcome: decisions on the priority types of TD for the team
TD DISCOVERY
Goal: to identify and document existing debts
Frequency: once, before introducing LTD
Outcome: inclusion of TD Stories in the backlog of the system
TD PLANNING
Goal: to select TD Stories to be repaid in a sprint
Frequency: recurrent, during sprint planning (or equivalent events)
Outcome: inclusion of TD Stories in the sprint backlog
TD PAYMENT
Goal: to repay the TD Stories includded in the Sprint Backlog
Frequency: recurrent, along with other development tasks
Outcome: reducing the debts existing in the system

Figure 6.1: Overview of the four activities proposed in LTD.

As discussed in Chapter 2, TD Management is a well-studied field in the literature. Previous studies proposed different methodologies to handle TD, regarding prevention strategies [Freire et al., 2020; Pérez et al., 2021; Freire et al., 2021], awareness raise [Besker et al., 2019; Eliasson et al., 2015; Rocha et al., 2017], and project management [Guo et al., 2016; Rios et al., 2020; Li et al., 2015]. By contrast, we argue that the previous proposed frameworks differ from LTD as (i) they focus on particular strategies to deal exclusively with TD (disregarding other activities that should be performed by developers); (ii) they do not take into account the singularities of different process methodologies (i.e., they are not flexible and customizable); (iii) they do not emphasize the adoption of a specific artifact to document TD (as we propose with the TD Stories, created with issues in tracking systems). In this chapter, we seek to answer the following research question:

RQ7. What are the perceptions of developers about LTD? In this RQ, we conduct a preliminary case study with two development teams from a large public company. They represent different development scenarios, in terms of development stages, deadline pressure, and technology stacks. In both, we conducted the activities proposed by LTD, observing the adoption of the framework and discussing its pros and cons with developers for one month.

6.2 LTD Framework

In order to support development teams to better document and to reduce Technical Debt, we propose LTD: Less Technical Debt framework. The primary goal of the framework is to introduce TD-related concerns into agile-based development methods, including activities that support developers to constantly document, prioritize and pay Technical Debt. To reach this goal, we propose that teams should: (i) reach a common understanding about TD, and its importance; (ii) identify and document TD stories; (iii) prioritize and plan TD payment; and (iv) dedicate a qualified time to pay TD Stories. In LTD, we propose that developers create issues in tracking systems to better document and manage TD (which we denote as TD Stories). As we showed in the previous chapters, this is a common practice among top-starred systems in GitHub (see Section 4.2). In Chapter 5, we also described the main circumstances in which the documentation of TD using issues is recommended.

In fact, LTD differs from previous frameworks as it intends to be (i) lightweight, with flexible and customizable activities; (ii) integrable with existing development processes (i.e., it is not intended to be an additional process that the team should master); and (iii) centered on the creation and management of TD Stories (i.e., issues in tracking systems). Therefore, we propose to inject four activities into agile-based development: TD Consensus, TD Discovery, TD Planning, and TD Payment. In this section, we detail LTD activities and provide examples about their adoption.

6.2.1 TD Consensus

Although the Technical Debt metaphor is intuitively known by developers, there is a lack of solid consensus about its actual definition and types [Kruchten et al., 2012; Rios et al., 2020; Ciolkowski et al., 2021]. For example, in Chapter 3, we showed that TD is related to 14 different types of concerns, varying from REQUIREMENTS to INFRASTRUCTURE. Therefore, the first activity proposed in LTD seeks to produce a common understanding among developers about TD concepts. Particularly, in this initial activity we propose that the team discuss the definitions of Technical Debt internalized by developers with the purpose of identifying possible misunderstandings that may affect the upcoming activities. It is also recommended that the team discuss the types of Technical Debt (e.g., DESIGN, TEST, BUILD, etc.), as well as the debts that they may consider most relevant to prioritize in the context of the software that they are developing. The results obtained in our characterization study (Chapter 3) can support this discussion and consensus. In practice, this activity has the goal of creating an essential agreement in the team to support the upcoming activities. It is also intended to promote awareness about TD, possibly preventing new debts.

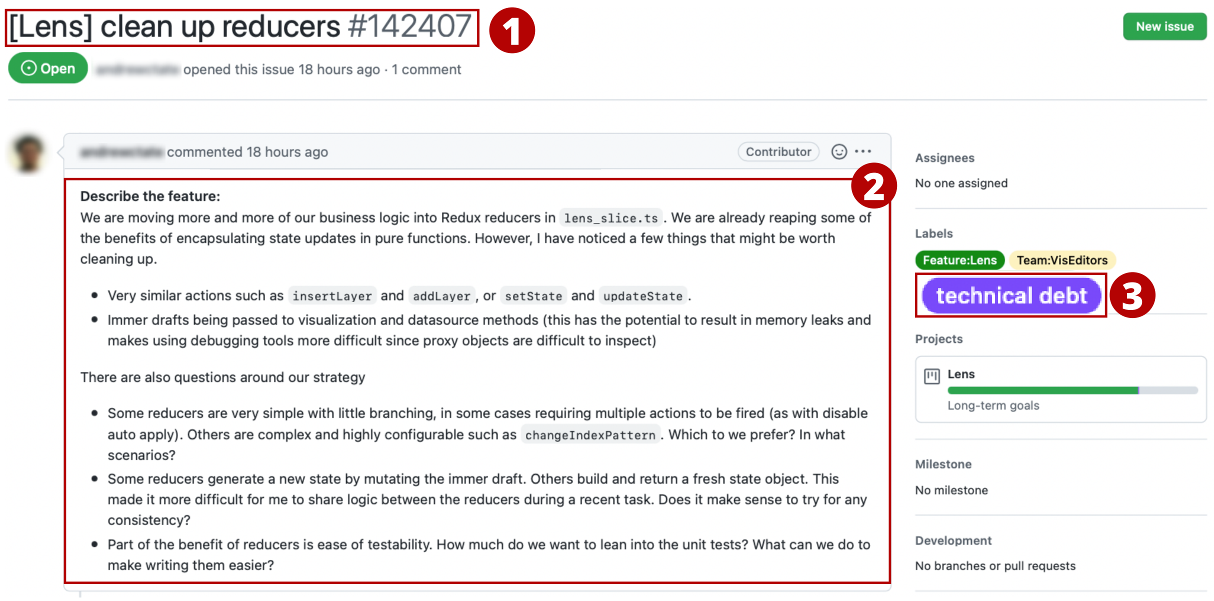
We acknowledge that TD CONSENSUS is not intended to be an activity that needs to be constantly conducted by teams (in every sprint, for example). By contrast, this activity is perhaps essential to happen in the initial phases of a project, or by the time LTD starts to be incorporated in the process. It may also be carried out in the form of workshops, webinars, or internal brainstorming. In methods that include an inception phase, it might be worthy including this discussion during the activities proposed at this stage of development. In any case, the project manager, scrum master, or any other external expert may lead the discussions.

As a result of this activity, it may also be relevant to document the definitions that the team reached out. For example, the team may decide to document and label TD in issue tracker systems, defining the meaning of these labels with the adopted concept. Although we cannot affirm that the repositories collected in Section 4.2 have conducted any activity equivalent to LTD's TD CONSENSUS, we observe that they adopt a common understanding about TD concept by defining the meaning of their TD-related label. In MICROSOFT/VSCODE, the team uses the label *debt* to document "Code quality issues". In INFLUXDATA/INFLUXDB, the label *kind/tech-debt* stands for issues that "Needs cleanup, will make the developer experience better".

6.2.2 TD Discovery

Based on the knowledge produced in TD Consensus, the second activity proposed by LTD aims to prospect Technical Debt instances. During this activity, developers are intended to elicit a catalog of **TD Stories**. We now denote as TD Stories the Satd-I instances studied in the previous chapters (i.e., during this activity, we recommend that developers document TD with issues in tracking systems). The rationale for switching from the term "Satd-I" to "TD Story" is to promote developer's understanding, as they tend to be more familiar with user stories, commonly adopted in agile methods.

Each TD Story should document and describe the debt, providing essential information that might be necessary for its payment. For example, Figure 6.2 illustrates an ideal TD Story, from elastic/kibana. As we can observe, the story includes a meaningful title (flag #1, in the figure), a not too long description (flag #2), and an indication of TD, with the *technical debt* label (flag #3).


[Lens] clean up reducers #142407 1

New issue

Open [redacted] opened this issue 18 hours ago · 1 comment

[redacted] commented 18 hours ago Contributor

2

Describe the feature:

We are moving more and more of our business logic into Redux reducers in `lens_slice.ts`. We are already reaping some of the benefits of encapsulating state updates in pure functions. However, I have noticed a few things that might be worth cleaning up.

- Very similar actions such as `insertLayer` and `addLayer`, or `setState` and `updateState`.
- Immer drafts being passed to visualization and datasource methods (this has the potential to result in memory leaks and makes using debugging tools more difficult since proxy objects are difficult to inspect)

There are also questions around our strategy

- Some reducers are very simple with little branching, in some cases requiring multiple actions to be fired (as with disable auto apply). Others are complex and highly configurable such as `changeIndexPattern`. Which to we prefer? In what scenarios?
- Some reducers generate a new state by mutating the immer draft. Others build and return a fresh state object. This made it more difficult for me to share logic between the reducers during a recent task. Does it make sense to try for any consistency?
- Part of the benefit of reducers is ease of testability. How much do we want to lean into the unit tests? What can we do to make writing them easier?

Assignees
No one assigned

Labels
Feature:Lens Team:VisEditors
technical debt 3

Projects
Lens
Long-term goals

Milestone
No milestone

Development
No branches or pull requests

Figure 6.2: Example of TD Story from elastic/kibana.

In practice, during TD Discovery the team should work to create a **TD Backlog**, which is essential to increase awareness, and to plan the payment of the debts. In fact, previous frameworks proposed in the literature also include the idea of such lists [Guo et al., 2016; Schmid, 2013]. However, in LTD we propose that the debts should be documented as stories, and the TD Backlog should be included in the same backlog managed by the team (i.e., for the sake of simplicity, it is recommended that the teams use a single and unique backlog. Of course, searching and filtering mechanisms can be easily used to select only the TD Stories). This may increase the visibility of the debts, as well as facilitate their management in parallel with other development activities. To this end, we recommend the creation of labeled issues, as described in the previous chapters. It is also recommended that developers prioritize the stories created, according to their relevance, complexity, or any other criteria decided by the team.

In relation to its execution, we claim that the first TD Discovery is an essential activity for ongoing teams that decide to adopt LTD after the beginning of the project. In this case, the implementation has started and it might be relevant to document already-existing debts. After the initial creation of the TD Backlog, it should be encouraged that developers keep the backlog updated, including the debts inserted in the codebase. In this scenario, it is essential the execution of an initial event, perhaps as a follow-up of the TD Consensus activity, in which the team should come up with the first version of the TD Backlog. In projects that decide to adopt LTD since its beginning, it is feasible that developers maintain the backlog as a daily activity (i.e., the initial discovery may not be relevant). In both cases, it is recommended that project managers or scrum masters include this discussion during sprint reviews, updating the backlog and discovering debts that developers might have forgotten to document during the sprint.

6.2.3 TD Planning

The ultimate goal of managing Technical Debt is to promote its payment. In this case, we claim that an essential key to achieve this goal is including this discussion during planning activities. To accomplish that, LTD proposes TD Planning, in which developers are intended to plan the inclusion of TD Stories in the sprint backlog (or any other artifact that describes the goal of the team during a given time box). In practice, during the planning activities already adopted by the team, it is recommended that developers select TD Stories with the highest priority that fit in the team capacity. Although LTD is intended to be flexible and integrable with any agile-based method, we acknowledge that this activity should be executed iteratively (e.g., during Sprint Planning events). In this case, developers may continuously reprioritize TD Stories, as well as plan their payment during a fixed time frame.

LTD also defines a percentage of the sprint that should be dedicated to TD Payment. In general, we recommend that the teams define around 20% of the sprint to pay Technical Debt. This threshold was inspired by the so-called "20% rule" used by some companies, such as Google. This rule allows developers to use 20% of their time to work on any project of their choice [Henderson, 2017]. For example, sprints of ten days should dedicate two days to pay Technical Debt. In practice, this time should be included as slack [Beck, 2000] or cool down activities [Singer, 2019]. In any case, it is essential that the team divide the team capacity considering this dedicated time, selecting TD Stories viable to be paid during this time. Finally, it is also worth mentioning that LTD is intended to be flexible. Thus, this percentage might be adjusted according to the team necessity or availability (e.g., projects in initial phases may not need the whole capacity).

6.2.4 TD Payment

Finally, the TD Payment activity refers to the days in which developers should concentrate effort to pay the TD Stories selected for the sprint. In this context, developers are intended to close the TD Stories opened in the backlog, refactoring suboptimal code or fixing workarounds previously identified. In this activity, it is important that project managers and scrum masters (or other equivalent leaders) guarantee that: (i) the teams cultivate a culture of actually carrying out the TD Stories (i.e., not postponing or giving low priority); (ii) clients or product owners agree that a portion of the sprint is dedicated to quality improvements (in this case, paying TD Stories). As a result, the team might cultivate an environment of recognizing the importance of such activities, mainly considering the future interest of the debts. Finally, it is important to permanently reinforce to the team that newly identified debts during TD Payment days should be included in the backlog in the form of a new TD Story.

6.3 Case Study Design

To explore the effectiveness of LTD, we conducted a case study with two distinct development teams from Prodemge, a large public company from Minas Gerais, Brazil. The company is responsible for developing and maintaining software solutions for more than 70 public clients from the state, such as: the state transport department (DETRAN-MG), the state gas company (GASMIG), and the educational and health councils (SEE and SES, respectively). Prodemge agreed to participate in an exploratory case study, in which we executed and evaluated LTD activities in two development teams, working for two distinct clients. Both teams adopt Scrum to develop their software solutions.

The first team (which we refer as Team 1) is composed of 14 employees, including one Product Owner, one Manager, one Scrum Master, two DevOps, two Software Architects, three Software Developers, one Requirements Analyst, one UX Intern and two Software Development Interns. Starting on November 30th, 2020, the goal of the team is developing software solutions to issue certificates for agricultural products, for identification of People with Autism Spectrum Disorder, and for declaration of days worked for people deprived of liberty. The main technologies adopted by the team are: Java, Jboss, Prodigio, ZK, Hibernate, EJB, Spring boot, and Oracle Database. In the second team (Team 2), 16 employees are dedicated to develop a solution that aims to manage information of legal actions in health. The team is composed of one Scrum Master, one Requirements Analyst, one Manager, one Supervisor, one Software Architect, one DevOps, four Software Developers, two Software Development Interns, one UX Engineer, one Product Owner and two Product Owner Assistants. The project started on July 10h, 2021 and is developed with PHP, Laravel, and Angular.

The execution of the study lasted one month. During this period, we conducted each LTD activity described in Section 6.2 and supported the teams along three sprints (two sprints with Team 1 and one sprint with Team 2). Figure 6.3 illustrates the adoption of LTD in both Teams and provides an example of the introduction of the framework in Scrum-based teams.

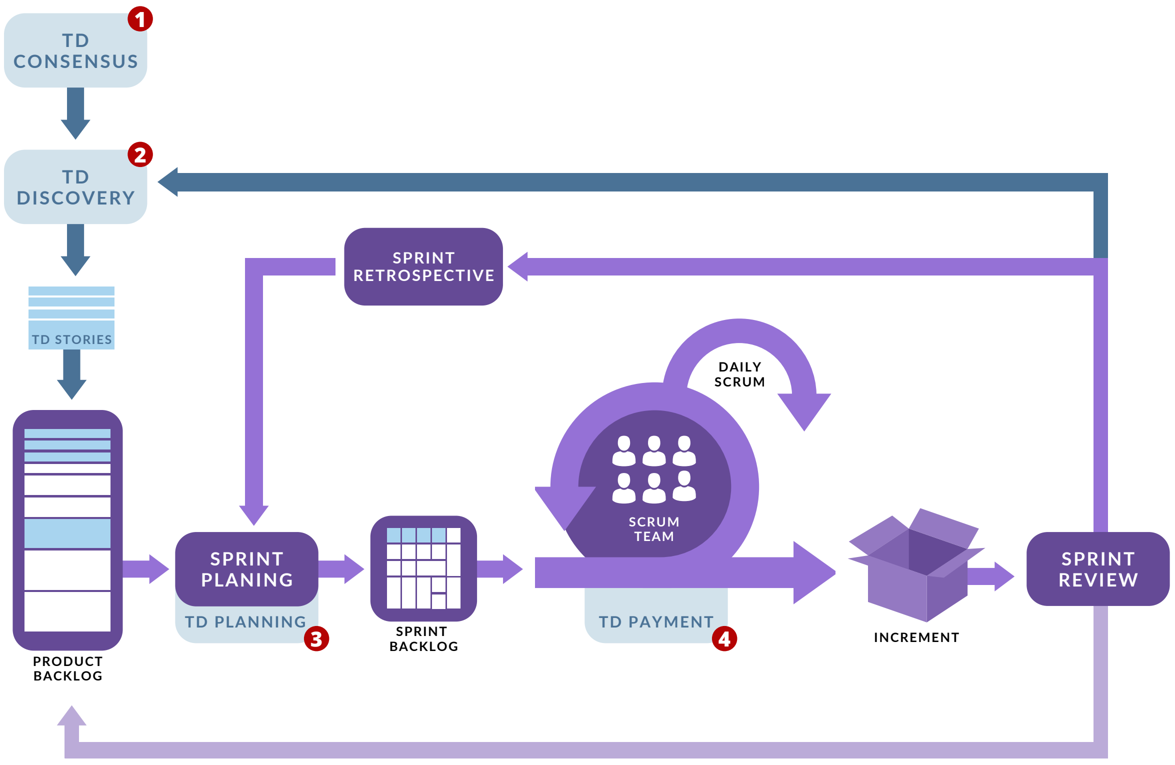


Figure 6.3: Example of LTD in Scrum-based teams (as conducted in the case study).

As we can observe in the figure, we conducted TD Consensus in the form of two online meetings (1.5 hour, each), one for each team (flag #1 in the figure). In these meetings, we started by discussing with the teams their initial knowledge on TD. Then, we presented the concept proposed by Cunningham [1992], detailing the most frequent types of debt (as presented in Section 3.3.1). We also provided examples to better support our definitions. The second part of the meetings were dedicated to explain LTD and propose the execution of TD Discovery. In this second activity (flag #2), the teams were invited to elicit TD Stories. As both projects started a few years ago, this activity was essential to include already existing TD issues into their backlogs. In both teams, the TD Discovery activities were performed asynchronously (i.e., the teams elicited TD Stories without any guidance, before the beginning of the upcoming sprint). Both teams started to adopt LTD in the sprint that began right after our first interactions. Therefore, we participated in the teams' Sprint Planning, in which they included the TD Planning activities (flag #3). During these meetings, we clarified the doubts about the TD Stories elicited and we also supported the prioritization of the ones selected for the sprint. During the sprint execution (flag #4), we participated in the teams' Daily Scrum meetings related to the TD Payment days (i.e., the meeting that happened in the days selected to pay TD). In the first team, the TD Payment days happened along the first three days of each sprint (this team adopted LTD in two sprints). Team 2 dedicated the last day of the sprint adopting LTD for paying the planned debts. During the Daily Scrums, we also clarified doubts and assessed developers' engagement on the activities. Finally, to gather information about the teams' perceptions about LTD and answer RQ7, we participated in their Sprint Retrospectives.

6.4 What are the perceptions of developers about LTD?

In this section, we answer RQ7 by reporting the experience of including LTD in two development teams from Prodemge. As described in Section 6.3, our study lasted one month, involving 30 employees working in three sprints (two sprints with Team 1 and one sprint with Team 2). Next, we describe the results achieved by the teams in each activity, as well as the perceptions collected about the adoption of the framework.

6.4.1 Team 1

Team 1 was the first to start adopting LTD, incorporating the proposed activities in two consecutive sprints. As described in Section 6.3, the team firstly participated in the TD Consensus activity, in which we discussed the concepts necessary to better identify and manage TD. During this meeting, we noticed that few members of the team were aware about the term "technical debt". The ones who have already heard about the metaphor, were not sure about its meaning. However, as the activity progressed, we also observed that the team was already familiar with the need of conducting maintenance activities to improve code quality. Indeed, they already had the practice of allocating an amount of time from the sprint to fix bugs, refactor code and improve documentation. However, the execution of these tasks did not follow a well-defined process of identification, prioritization and planning. By the end of the TD Consensus, the team defined strategies to document and maintain TD Stories. Particularly, they decided to document the stories into the product backlog of the project, including issues marked with a TD-related label (i.e., "dívida técnica", in Portuguese). Therefore, the stories were incorporated into their repository's project board in GitLab, as recommended in LTD (based on the practice described in Sections 3.2 and 4.2).

The TD Discovery activity was conducted asynchronously, before the beginning of the upcoming sprint. In this activity, the team elicited and documented six TD Stories. Furthermore, 16 additional stories were included in the backlog during the study. They were mainly identified while developers were paying the initially identified debts, discussing payment strategies, or implementing other development tasks planned in the sprint. Table 6.1 describes the final backlog of TD Stories documented by Team 1, including their description, the type of TD (as classified in Section 3.3.1), the status of the issue (closed or open), and the event when it was paid (the first or the second sprint in which the studied was performed). The table is divided by the event when the debt was identified (i.e., during TD Discovery, Sprint #1, and Sprint #2, respectively).

As we can observe in Table 6.1, the team identified more TD Stories after TD Discovery (6 stories in TD Discovery vs 16 during the sprints). This might be explained by the fact that the team was not familiar with the concept in the beginning of the study. Therefore, during LTD implementation, it might be worthwhile to cultivate a culture of discussing TD items during the execution of the sprint, as well as keeping the backlog open so that developers may create new stories while working on payment activities. Moreover, we also claim that Team 1 mostly documented debts related to Design, UI and Build (as classified in Section 3.3). For example, the story #14 refers to the need of refactoring the requests of the system to create a common approach to pass parameters. Therefore, it is classified as Design debt, since it refers to technical

Table 6.1: Final backlog of TD Stories documented by Team 1.

TD Story	**Type**	**Status**	**Paid**
1. Define ZK upload components for Portuguese	UI	Closed	SP.1
2. Migrate chosenbox components to the theme class	DESIGN	Closed	SP.1
3. Configure chosenbox component classes	DESIGN	Closed	SP.1
4. Create the radio component classes in the theme class	UI	Closed	SP.1
5. Define a standardized way of passing parameters in requests	ARCHITECTURE	Closed	SP.1
6. Change the way to interpret the components when it occupies 100% of the line	UI	Closed	SP.1
7. Migrate toggle components to the theme class	UI	Closed	SP.2
8. Configure toggle component classes	UI	Closed	SP.2
9. Migrate colorbox components to the theme class	DESIGN	Closed	SP.2
10. Configure colorbox component classes	UI	Closed	SP.2
11. Treat uppercase fields by the theme's css and not by zul	UI	Closed	SP.2
12. Define a group in git and change the API to accept group configuration	BUILD	Closed	SP.2
13. Request creation of a user for git integration and change git API to accept user configuration	BUILD	Closed	SP.2
14. Implement a standardized way of passing parameters in requests	ARCHITECTURE	Open	-
15. Isolate the implementation of communication with git in a microservice	BUILD	Open	-
16. Create the project for CI/CD	INFRASTRUCTURE	Open	-
17. Configure the publishing material and properties	DESIGN	Open	-
18. Create the repository in GIT for automation prototype	INFRASTRUCTURE	Open	-
19. Make the log available in graylog	ARCHITECTURE	Open	-
20. Migrate auto-publishing logic	ARCHITECTURE	Open	-
21. CSS from radio to theme	UI	Open	-
22. CSS from buttons without btn class	UI	Open	-

shortcuts used when defining high-level architecture.

In both sprints that we supported Team 1 to incorporate LTD, we participated in the team's Sprint Planning meetings. During these events, the team estimated and prioritized the stories remaining open in the backlog. For example, before the first sprint adopting LTD, the backlog included the six stories identified during TD Discovery (as detailed in Table 6.1). During the sprint planning, they decided to include all of them in the sprint, since the time estimated to pay the stories was compatible with the 20% time window. In both sprints, the team decided to follow our recommendation of starting the sprint with TD Payment days. Therefore, we participated in four Daily Scrum meetings (two in each sprint), in which we observed that the team engaged in the activities of paying debts, as well as incorporated the culture of documenting new debts in the backlog. In some cases, the team also needed support to classify and identify new debts (e.g., a common doubt was related to what items were actual debts or actually new feature requests).

In the end, we participated in the team's Sprint Retrospective meetings to observe their impressions about the framework. In both meetings, the team highlighted two main elements of LTD: the creation of TD Stories and the allocation of time for exclusively paying TD. In relation to the documentation of the debts, the team discussed the relevance of having a common artifact in which they could concentrate the debts identified. By adopting such practice, they acknowledged that it was easier to remember the existence of the concerns, increasing the chances of fixing the workarounds. In fact, this practical observation is in accordance with the literature that also discusses the importance of TD awareness [Besker et al., 2019; Eliasson et al., 2015]. Equally, the team also discussed the relevance of allocating a fixed time-frame for paying TD. In this case, they compared this strategy with their previous practice of including time for code improvements. In this case, they argued that LTD was able to promote the improvement of artifacts that were not commonly considered in this initial practice (i.e., artifacts not related to code, e.g., documentation), and created a structured approach to reduce the debts.

As a result, we observed that Team 1 paid 59% of the debts identified during the study (13 TD Stories closed, among the 22 included in the backlog). Moreover, they could manage to pay 100% of the debts planned in each sprint, as detailed in Table 6.1. Therefore, we claim that LTD was effective to reduce TD in Team 1.

6.4.2 Team 2

The second team to adopt LTD at Prodemge included the activities proposed in the framework for one sprint. Similarly to Team 1, we started our case study with Team 2 by supporting the execution of the TD Consensus activity. In this team, we perceived that more developers were familiar with the metaphor and its concept, although they claimed that they have never adopted any process to manage TD. By contrast, the team seemed to be caught by the different types of TD. Initially, they had a previous impression that the concept was restricted to code workarounds. By the end of this first activity, the team decided to document TD stories in the form of Trello cards, labeling the cards with a TD-related tag (i.e., "dívida técnica", in Portuguese). Similarly to GitHub or GitLab Projects, Trello also allows the team to create backlogs, with several cards. Each card can include a title, a description, its assignees, comments and due date. For this team, the cards are used to represent the stories that they should implement in the sprint. As for Team 1, Team 2 also followed LTD instructions and decided to include TD Stories into their usual backlog.

Because this team adopted LTD for one single sprint, the identification of TD Stories was restricted to the TD Discovery activity. For that, the team also conducted this activity asynchronously, creating TD Stories before the beginning of the upcoming sprint. As a result, Team 2 elicited seven TD Stories. Table 6.2 details the description of these stories, as well as the type of TD (as classified in Section 3.3), its final status and the event when they were paid. This table is not divided by the event when the debts were identified because all of them were discovered during TD Consensus. As we can see, this team mostly identified debts related to Design, Architecture, and UI. For example, the TD Story #3 refers to the necessity of fixing a workaround in which the relationships in the code architecture are not mapped to the database entities' relationships. In this case, the database was not created to reflect the structure of the relationships in code.

Table 6.2: Final backlog of TD Stories documented by Team 2.

TD Story	**Type**	**Status**	**Paid**
1. Document API endpoints	Documentation	Open	-
2. Conclude the integration with WSO2	Architecture	Ongoing	-
3. Create relationships on foreign keys in JUD tables	Architecture	Ongoing	-
4. Improve return of users type (Group and Level)	Design	Closed	SP.1
5. Improve return of users type (Health Establishment)	Design	Closed	SP.1
6. Refactor User registration	Design	Closed	SP.1
7. Fix interface to search unit of measure (+ sign)	UI	Closed	SP.1

One particularity of the team refers to its current scenario of development. In contrast to Team 1, they were under pressure to deliver important features. Because of that, during the TD Planning activities, the team could not allocate 20% of their capacity to pay TD. Instead, the team planned to dedicate 10% of the sprint to pay the most priority debts identified during TD Discovery. Therefore, they planned to pay six stories (leaving story #1 for a future sprint). Even though, during the TD Payment days, the team was surprised with urgent demands that resulted in the delay of the payment of two stories (i.e., #2 and #3). Finally, during TD Payment the team could conclude four TD Stories, and left three for future sprints. Therefore, the team paid almost 67% of the debts identified during TD Discovery.

By observing the team's Sprint Retrospective, we could perceive that the inclusion of LTD into the team process of development was relevant to increase the awareness about the debts existing in the project. Besides, it was common to observe the team discussing strategies to avoid TD during the implementation of other stories, which represents an important progress to prevent future debts. During the retrospective, the team also highlighted the importance of documenting the debts along with other stories of the project, creating an important reminder of workarounds that need to be fixed. Moreover, the team suggested the importance of creating an agreement at the business level (with clients and product owners), so that the 20% time frame could be effectively allocated. Even though we claim that LTD was effective to reduce TD in Team 2.

6.5 Implications

Based on our results and the perceptions observed during the study, we shed light on the following practical implications:

1. LTD can be effective to reduce TD. Both of the teams that participated in this exploratory study closed a relevant percentage of the debts documented during TD Discovery (59% in Team 1 and 67% in Team 2). We highlight that Team 1 and Team 2 were experiencing distinct scenarios of development during the study. Team 2 was under pressure to deliver important features. Even thought, this context did not impact in our observations (in terms of percentage of paid debts). Moreover, we also claim that the obtained results were not influenced by previous knowledge and practices of the team. As discussed in Section 6.4, Team 1 used to conduct an informal activity of code improvement before the study, which was not restricted to managing TD. By contrast, developers in Team 2 were less aware about the concept. Even thought, we observe that the obtained results were equally satisfactory.

2. LTD does not require extra-effort of the teams. Besides the time needed to conduct TD Consensus and TD Discovery—which are necessary for new adopters of the framework—we highlight that no extra-effort was required to inject LTD activities in the teams' development processes. Particularly, after the conduction of the aforementioned activities, both teams included TD Planning and TD Payment as part of their usual Scrum activities. Moreover, as the debts were documented in the form of TD Stories, included in their backlogs, their payment was dealt by developers as a regular activity of the sprint. In comparison to previous studies in the literature, we claim that this light-weight characteristic of LTD may facilitate its adoption, as the team is not required to follow extra workflows.

3. LTD promotes awareness about TD. The adoption of LTD was also responsible for including TD as a subject of discussions in the teams, which contributed to the identification of more TD Stories. We claim that this implication may be a consequence of two relevant aspects of the framework: (i) the TD Consensus activity is indeed a relevant event for the alignment of a team; (ii) the existence of TD Stories in the backlog calls attention about the necessity of dealing with this concern. Particularly, the latter aspect was also discussed by developers in Chapter 5 (i.e., 25% of the surveyed developers recommended documenting TD in issues to foster its visibility). As previously discussed in the literature, TD awareness is also a key for preventing the inclusion of new TD items, which can be measured in these teams as future work.

6.6 Threats to Validity

First, our exploratory study was restricted to two development teams, from one public software company. Although we selected teams in distinct scenarios of development, adopting different stacks of technology, from a large software company, we claim that our results can not be generalized. Besides, we only included LTD in Scrum-based teams. Therefore, the results of its adoption in different methodologies is still unclear. Next, our study lasted one month, including two sprints of Team 1 and one sprint of Team 2. Because of that, our results were restricted to analyzing the amount of debts paid by the teams, during this period. The long-term impact of the adoption of LTD is also a threat to validity to our conclusions. Finally, several considerations included in this chapter are fruit of the observation of the author of this thesis. However, we claim that this is a natural path in this kind of study and that a critical point of view was also assumed by the author, while analyzing the results.

6.7 Final Remarks

In this chapter, we proposed and evaluated LTD: Less Technical Debt Framework. With the purpose of supporting developers to better document and manage TD, the framework proposes the inclusion of four lightweight activities into the development process adopted by a team. They are:

- TD Consensus: in this activity, the team seeks to create a common understanding about the Technical Debt metaphor and its types. Besides, it is also expected that they agree with the most relevant TD types for the team and decide the strategies that will be adopted to document TD Stories (i.e., TD items present in the project).
- TD Discovery: following the TD Consensus, in this activity the team aims to identify the TD Stories already existing in the system, including such debts in the project's backlog.
- TD Planning: before the execution of a development cycle, during its planning, the team is also expected to decide the most priority TD Stories that can be paid. For that, LTD suggests that the team allocates 20% of its capacity to dedicate for TD payment.
- TD Payment: finally, the team is expected to pay the planned TD Stories during the days allocated for this purpose. Particularly, LTD suggests that TD Payment days should be carried out in the beginning of the development cycle.

To evaluate the effectiveness of LTD, we conducted a case study with two development teams from Prodemge (a large public company from Minas Gerais, Brazil). The teams adopted LTD during three sprints and carried out each proposed activity with our support and guidance. As a result, we observed the payment of 13 TD Stories in Team 1 and four in Team 2. Besides, the teams' backlog also included 12 additional TD Stories identified during the study. By observing the execution of the activities, we could also perceive the teams' engagement to pay the identified debts, highlighting the importance of (i) creating a common artifact in which TD is documented; (ii) allocating a fixed time frame to pay TD; (iii) including TD concerns among the team discussions and decisions. As a next step, we aim to explore the inclusion of LTD in additional teams, from different companies, facing different development scenarios. Besides, we also plan to explore strategies to prioritize and estimate TD Stories.

Chapter 7

Conclusion

In this chapter, we first describe the four major works conducted throughout this thesis (Section 7.1). Next, in Section 7.2 we list the main contributions of these efforts. Finally, Section 7.3 outlines the future work that we find interesting for follow-up researches.

7.1 Thesis Recapitulation

The Technical Debt (TD) metaphor frames a common practice in modern software development: to increase productivity and release earlier, developers usually compromise the internal quality of the software by taking suboptimal solutions. Indeed, in the short-term, TD may contribute to faster feedbacks and to the rapid growth of the software. However, in the long-term, the cost of TD accumulation can be devastating. To deal with this problem, developers usually document their decisions, which the literature denotes as Self-Admitted Technical Debt (SATD). Most previous studies investigated this documentation strategy by analyzing source code comments. In this case, they concentrate efforts in mining indications of TD in developers comments, e.g., by searching *TODO*, *Fixme*, or *Hack* keywords. We denote this practice as SATD-C. By contrast, few previous studies prospected TD indications in issue tracker systems (which we refer as SATD-I). In this context, they analyzed very particular excerpts of issues (e.g., title and comments) to explore this practice. We report in this thesis a set of four major work units where we extensively analyzed SATD-I, investigating its adoption as means to document and manage Technical Debt.

We started by defining what Technical Debt is, with particular focus on SATD (Chapter 2). We also discussed the most common techniques proposed in the literature to identify, manage, and pay SATD, mostly regarding the ones documented in source code comments. Finally, we described the state-of-the-art concerning studies related to SATD-C and SATD-I.

Next, we reported in Chapter 3 our first exploratory study, in which **we empir-**

ically characterized the adoption of issues as means to document Satd. For that, we first collected an initial dataset with 286 issues tagged with a TD-related label (which we consider as a proxy of TD admission). Based on this dataset, we investigated the main types of debts documented in Satd-I (e.g., debts related to Design, Tests, or Documentation flaws). We also surveyed developers involved in Satd-I payment to assess the circumstances of the creation and payment of such debts.

In Chapter 4, we strengthen our first work by investigating the interplay between debts reported in code comments and in issues. Particularly, we assumed that issues are indeed adopted to document TD (Chapter 3) and concentrated our efforts to understand the relationship between Satd-C and Satd-I. For that, we implemented a prototype tool called AdmiTD, whose goal is to automatically transform Satd-C comments in GitHub issues. To assess the interest in such transformations, we surveyed the core developers of 10 GitHub repositories. Moreover, we also investigated whether it is a common practice to link both forms of Satd (i.e., to include references to Satd-I in Satd-C comments). The existence of such references would indicate the feasibility of creating a tool to automatically identify and link debts reported in both forms of Satd. In summary, **we concluded in this second study that Satd-C and Satd-I are adopted to document TD in distinct situations.**

Next, we dedicated Chapter 5 to explore the circumstances that drive developers to choose between one or another. For that, we identified and surveyed developers who reported Satd using both strategies (i.e., the ones who created both Satd-C comments and Satd-I issues). As a result, **we elicited a catalog of recommendations to support developers to better document Satd.**

Finally, in Chapter 6 we wrapped up the knowledge gained with our previous studies and explored the effectiveness of adopting issues in activities to manage TD. For that, **we proposed the Less Technical Debt framework (LTD):** a customizable and lightweight framework that aims to inject TD concerns in agile-based methodologies. Specifically, the framework defines four main activities: TD Consensus, to create a common understanding among developers about TD concepts; TD Discovery, to prospect and document the TD already existing in the system; TD Planning, to include the payment of TD into the activities planned by the team; and TD Payment, to define days in a sprint that the team should pay TD. **We also assessed LTD in a real-world scenario, involving 30 stakeholders and two development teams.**

7.2 Contributions

In the context of the research conducted in this thesis, we highlight the following contributions:

- We initially confirmed that developers use issues to admit Technical Debt in their projects, i.e., SATD does not appear only in code (Chapter 3). We also showed that the majority of the debts documented in issues are related to DESIGN flaws (60%), followed by UI, TESTS, and PERFORMANCE concerns. We also provided insights about the reasons of SATD-I creation and payment, indicating that almost 45% of the studied debts were introduced to ship earlier and almost 65% are paid to reduce their interest.
- We created a large-scale dataset that includes 20K SATD-I issues and 72K SATD-C comments.[1] To the best of our knowledge, it is the largest dataset containing both forms of SATD. We also implemented a prototype tool that automatically transforms SATD-C comments in GitHub issues (SATD-I).[2] However, we showed that there is a negligible interest in such transformations. Moreover, we also showed that linking SATD-C and SATD-I is not a common practice among developers. Ultimately, we concluded that there is a minor interplay between both forms of SATD (Chapter 4).
- We provided a catalog with 13 guidelines to support developers to choose between code comments and issues when documenting SATD (Chapter 5). The catalog includes six recommendations to use SATD-C, including the circumstances when developers aim to provide context to the reader (58%), to report low priority debts (24%), and to document local-scoped TD (19%). By contrast, developers mostly use issues to foster discussion with other team members (31%), to document TD that needs to be tracked (27%), and to report debts that span in multiple places (25%). Moreover, the proposed catalog was partially included in the wiki documentation of an open-source repository (COCKROACHDB/COCKROACH).
- We proposed a lightweight framework that promotes TD management and documentation through the adoption of issues (Chapter 6). We denote the framework as LTD, for Less Technical Debt framework. We showed that LTD was indeed effective to reduce TD in the teams that participated in our case study in a large public software company, creating awareness and stimulating a culture of constantly documenting and paying debts.

[1] `https://doi.org/10.5281/zenodo.6532378`
[2] `https://github.com/admitd`

7.3 Future Work

During the works conducted in this thesis, we identified some unexplored questions that can result in relevant studies. We enumerate these future works in the following topics:

Automatically Labelling SATD-I: Although recent studies explored the automatic recommendation of labels for GitHub issues, to the best of our knowledge, there is a lack of studies that rely on such techniques in the Technical Debt field. Indeed, as discussed in Chapter 2, SATD-I is barely studied in the literature. Therefore, we claim that future work may include efforts with the goal of proposing and evaluating techniques responsible for identifying unlabeled SATD-I: issues that document TD concerns but do not include any TD-related label. In this case, such studies may evaluate (i) the extent of this problem (i.e., to answer the question: what is the frequency of unlabeled SATD-I?); and (ii) the benefits of such labels in terms of time to resolve the issue or developers engagement.

Catalog of SATD-I Payment Strategies: Despite the evident interest in the literature on SATD payment, we claim that there is a lack of understanding about this activity under the SATD-I perspective. As observed in Chapter 5, developers usually recommend the adoption of issues to document debts with global scope and high priority concerns, which may require different/more complex code changes to be paid. Thus, we suggest as future work the conduction of studies with the purpose of cataloging the code changes performed by developers to pay SATD-I. Moreover, we highlight that such studies may focus on DESIGN debts—the most common type of SATD-I—to understand *how* developers effectively pay such debts. For that, it is possible to rely on the closed issues in our dataset (Section 4.2), qualitatively analyzing the commits associated to their closing event (i.e., the `closed_by` field in GitHub issues). For example, such studies may analyze the commit messages provided by developers, as well as the associated code diff. As a result, this analysis may reveal some patterns that might extend the catalog presented by Zampetti et al. [2018].

Expand LTD Assessment: In Chapter 6, we assessed LTD with two development teams, from a large public company from Minas Gerais (Brazil). Although we selected teams in distinct stages of development, working on systems from different domains, and adopting distinct technologies, it is worthwhile assessing the adoption of the framework in other scenarios. For example, future studies may investigate the adoption of LTD in teams adopting a development process different from Scrum (e.g., Lean, XP, or Kanban). Moreover, it might be relevant to use and evaluate LTD activities in private companies, or exploring new contexts (e.g., startups or newly created projects). In such cases, it would be interesting to investigate the adoption of the framework since the beginning of the development process, and the effects of managing TD right after the first decisions of the project.

Explore Strategies to Prioritize and Estimate TD Stories: As discussed in Chapter 6, it is also relevant to explore strategies to support developers in prioritizing and estimating the effort needed to pay TD Stories. Particularly, during the TD Planning activities, we observed that developers commonly discussed the most relevant stories for a following sprint. In such cases, it could be relevant to include a strategy to highlight this information during TD Discovery activities (e.g., by including a priority label in the created stories). Moreover, considering that LTD relies on a fixed time frame to pay TD, we also highlight the importance of exploring strategies to help developers to better estimate the effort needed to pay a debt.

Bibliography

Alonso-Abad, J. M., López-Nozal, C., Maudes-Raedo, J. M., and Marticorena-Sánchez, R. Label prediction on issue tracking systems using text mining. *Progress in Artificial Intelligence*, 8:325–342, 2019.

Alves, N. S., Ribeiro, L. F., Caires, V., Mendes, T. S., and Spínola, R. O. Towards an ontology of terms on technical debt. In *6th International Workshop on Managing Technical Debt (MTD)*, pages 1–7, 2014.

Alves, N. S., Mendes, T. S., de Mendonça, M. G., Spínola, R. O., Shull, F., and Seaman, C. Identification and management of technical debt. *Information and Software Technology*, 70(C):100–121, 2016.

Anvik, J., Hiew, L., and Murphy, G. C. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE)*, pages 361–370, 2006.

Avgeriou, P., Kruchten, P., Ozkaya, I., and Seaman, C. Managing technical debt in software engineering. In *Dagstuhl reports*, volume 6, 2016.

Azuma, H., Matsumoto, S., Kamei, Y., and Kusumoto, S. An empirical study on self-admitted technical debt in dockerfiles. *Empirical Software Engineering*, 27:1–26, 2022.

Bavota, G. and Russo, B. A large-scale empirical study on self-admitted technical debt. In *13th Working Conference on Mining Software Repositories (MSR)*, pages 315–326, 2016.

Beck, K. *Extreme programming explained: embrace change.* Addison-Wesley Professional, 2000.

Bellomo, S., Nord, R. L., Ozkaya, I., and Popeck, M. Got technical debt? Surfacing elusive technical debt in issue trackers. In *13th International Conference on Mining Software Repositories (MSR)*, pages 327–338, 2016.

Besker, T., Martini, A., and Bosch, J. Technical debt triage in backlog management. In *2nd International Conference on Technical Debt (TechDebt)*, pages 13–22, 2019.

Bettenburg, N., Premraj, R., Zimmermann, T., and Kim, S. Duplicate bug reports considered harmful... really? In *24th International Conference on Software Maintenance (ICSME)*, pages 337–345, 2008.

Bissyandé, T. F., Lo, D., Jiang, L., Réveillère, L., Klein, J., and Traon, Y. L. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 188–197, 2013.

Borges, H., Hora, A., and Valente, M. T. Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334–344, 2016.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 255–265, 2018.

Brito, A., Valente, M. T., Xavier, L., and Hora, A. You broke my code: Understanding the motivations for breaking changes in APIs. *Empirical Software Engineering*, 25: 1458–1492, 2020.

Cabot, J., Cánovas Izquierdo, J. L., Cosentino, V., and Rolandi, B. Exploring the use of labels to categorize issues in open-source software projects. In *22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 550–554, 2015.

Camilo, F., Meneely, A., and Nagappan, M. Do bugs foreshadow vulnerabilities? a study of the chromium project. In *2th Working Conference on Mining Software Repositories (MSR)*, pages 269–279, 2015.

Choetkiertikul, M., Dam, H. K., Tran, T., and Ghose, A. Characterization and prediction of issue-related risks in software projects. In *12th Working Conference on Mining Software Repositories (MSR)*, pages 280–291, 2015.

Ciolkowski, M., Lenarduzzi, V., and Martini, A. 10 years of technical debt research and practice: Past, present, and future. *IEEE Software*, 38(6):24–29, 2021.

Cunningham, W. The WyCash portfolio management system. In *7th Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 29–30, 1992.

da Fonseca Lage, L. C., Kalinowski, M., Trevisan, D., and Spinola, R. Usability technical debt in software projects: A multi-case study. In *13th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–6, 2019.

Dai, K. and Kruchten, P. Detecting technical debt through issue trackers. In *5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*, pages 59–65, 2017.

de Freitas Farias, M. A., de Mendonça Neto, M. G., Kalinowski, M., and Spínola, R. O. Identifying self-admitted technical debt through code comment analysis with a contextualized vocabulary. *Information and Software Technology*, 121:106270–106270, 2020.

Eliasson, U., Martini, A., Kaufmann, R., and Odeh, S. Identifying and visualizing architectural debt and its efficiency interest in the automotive domain: A case study. In *7th International Workshop on Managing Technical Debt (MTD)*, pages 33–40, 2015.

Ernst, N. A., Bellomo, S., Ozkaya, I., Nord, R. L., and Gorton, I. Measure it? Manage it? Ignore it? Software practitioners and technical debt. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*, pages 50–60, 2015.

Fahid, F. M., Yu, Z., and Menzies, T. Better technical debt detection via SURVEYing. *Computing Research Repository*, abs/1905.08297, 2019.

Farias, M. A., Santos, J. A., Kalinowski, M., Mendonça, M., and Spinola, R. O. Investigating the identification of technical debt through code comment analysis. In *18th International Conference on Enterprise Information Systems (ICEIS)*, pages 284–309, 2016.

Flisar, J. and Podgorelec, V. Identification of self-admitted technical debt using enhanced feature selection based on word embedding. *IEEE Access*, 7(1):106475–106494, 2019.

Fontana, F. A., Ferme, V., and Spinelli, S. Investigating the impact of code smells debt on quality code evaluation. In *3rd International Workshop on Managing Technical Debt (MTD)*, page 15–22, 2012.

Fowler, M. Technicaldebtquadrant. `https://martinfowler.com/bliki/TechnicalDebtQuadrant.html`, 2009. [accessed 10-October-2019].

Freire, S., Rios, N., Gutierrez, B., Torres, D., Mendonça, M., Izurieta, C., Seaman, C., and Spínola, R. O. Surveying software practitioners on technical debt payment practices and reasons for not paying off debt items. In *24th Evaluation and Assessment in Software Engineering (EASE)*, pages 210–219, 2020.

Freire, S., Rios, N., Pérez, B., Castellanos, C., Correal, D., Ramač, R., Mandić, V., Taušan, N., López, G., Pacheco, A., Falessi, D., Mendonça, M., Izurieta, C., Seaman, C., and Spínola, R. How experience impacts practitioners' perception of causes and effects of technical debt. In *13th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 21–30, 2021.

Fucci, G., Zampetti, F., Serebrenik, A., and Penta, M. D. Who (self) admits technical debt? In *36th International Conference on Software Maintenance and Evolution (ICSME)*, pages 672–676, 2020.

Fucci, G., Cassee, N. W., Zampetti, F., Novielli, N., Serebrenik, A., and Penta, M. D. Waiting around or job half-done? Sentiment in self-admitted technical debt. In *18th International Conference on Mining Software Repositories (MSR)*, pages 1–10, 2021.

Guo, Y., Spínola, R. O., and Seaman, C. Exploring the costs of technical debt management–a case study. *Empirical Software Engineering*, 21(1):159–182, 2016.

Guo, Z., Liu, S., Liu, J., Li, Y., Chen, L., Lu, H., and Zhou, Y. How far have we progressed in identifying self-admitted technical debts? a comprehensive empirical study. *ACM Transactions on Software Engineering Methodology*, 30:1–56, 2021.

Haki, K., Rieder, A., Buchmann, L., and W.Schneider, A. Digital nudging for technical debt management at credit suisse. *European Journal of Information Systems*, 0(0): 1–17, 2022.

Henderson, F. Software engineering at google. *arXiv preprint arXiv:1702.01715*, 2017.

Huang, Q., Shihab, E., Xia, X., Lo, D., and Li, S. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering*, 23(1): 418–451, 2018.

Iammarino, M., Zampetti, F., Aversano, L., and Penta, M. D. Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In *35th International Conference on Software Maintenance and Evolution (ICSME)*, pages 186–190, 2019.

Iammarino, M., Zampetti, F., Aversano, L., and Di Penta, M. An empirical study on the co-occurrence between refactoring actions and self-admitted technical debt removal. *Journal of Systems and Software*, 178:110976–110976, 2021.

Kallis, R., Sorbo, A. D., Canfora, G., and Panichella, S. Ticket Tagger: Machine learning driven issue classification. In *35th International Conference on Software Maintenance and Evolution (ICSME)*, pages 406–409, 2019.

Kamei, Y., Maldonado, E. D. S., Shihab, E., and Ubayashi, N. Using analytics to quantify the interest of self-admitted technical debt. In *1st International Workshop on Technical Debt Analytics (TDA)*, pages 68–71, 2016.

Kashiwa, Y., Nishikawa, R., Kamei, Y., Kondo, M., Shihab, E., Sato, R., and Ubayashi, N. An empirical study on self-admitted technical debt in modern code review. *Information and Software Technology*, 146:106855–106855, 2022.

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

Li, Y., Soliman, O., and Avgeriou, P. Identification and remediation of self-admitted technical debt in issue trackers. In *46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 495–503, 2020.

Li, Y., Soliman, M., and Avgeriou, P. Identifying self-admitted technical debt in issue tracking systems using machine learning. *Empirical Software Engineering*, 1:1–1, 02 2022.

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

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

Liu, Z., Huang, Q., Xia, X., Shihab, E., Lo, D., and Li, S. SATD Detector: a text-mining-based self-admitted technical debt detection tool. In *40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 9–12, 2018.

Maipradit, R., Lin, B., Nagy, C., Bavota, G., Lanza, M., Hata, H., and Matsumoto, K. Automated identification of on-hold self-admitted technical debt. In *20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 54–64, 2020a.

Maipradit, R., Treude, C., Hata, H., and Matsumoto, K. Wait for it: Identifying "on-hold" self-admitted technical debt. *Empirical Software Engineering*, 25:3770–3798, 2020b.

Maldonado, E. D. S. and Shihab, E. Detecting and quantifying different types of self-admitted technical debt. In *7th International Workshop on Managing Technical Debt (MTD)*, pages 9–15, 2015.

Maldonado, E. D. S., Abdalkareem, R., Shihab, E., and Serebrenik, A. An empirical study on the removal of self-admitted technical debt. In *33rd International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248, 2017a.

Maldonado, E. D. S., Shihab, E., and Tsantalis, N. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering*, 43(11):1044–1062, 2017b.

Martini, A., Besker, T., and Bosch, J. Technical debt tracking: Current state of practice: A survey and multiple case study in 15 large organizations. *Science of Computer Programming*, 163:42–61, 2018.

McConnell, S. Technical debt. *Software Best Practices*, 2007.

Potdar, A. and Shihab, E. An exploratory study on self-admitted technical debt. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 91–100, 2014.

Pérez, B., Castellanos, C., Correal, D., Rios, N., Freire, S., Spínola, R., Seaman, C., and Izurieta, C. Technical debt payment and prevention through the lenses of software architects. *Information and Software Technology*, 140:106692, 2021.

Ramasubbu, N. and Kemerer, C. Integrating technical debt management and software quality management processes: a framework and field tests. In *40th International Conference on Software Engineering (ICSE)*, pages 883–883, 2018.

Ren, X., Xing, Z., Xia, X., Lo, D., Wang, X., and Grundy, J. Neural network-based detection of self-admitted technical debt: from performance to explainability. *ACM Transactions on Software Engineering and Methodology*, 28(3):1–45, 2019.

Rios, N., de Mendonça Neto, M. G., and Spínola, R. O. A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. *Information and Software Technology*, 102(1):117–145, 2018.

Rios, N., Spínola, R. O., Mendonça, M., and Seaman, C. The practitioners' point of view on the concept of technical debt and its causes and consequences: a design for a global family of industrial surveys and its first results from brazil. *Empirical Software Engineering*, 25(5):3216–3287, 2020.

Rocha, H., Valente, M. T., Marques-Neto, H., and Murphy, G. An empirical study on recommendations of similar bugs. In *23rd International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 46–56, 2016.

Rocha, J. C., Zapalowski, V., and Nunes, I. Understanding technical debt at the code level from the perspective of software developers. In *31st Brazilian Symposium on Software Engineering*, pages 64–73, 2017.

Schmid, K. A formal approach to technical debt decision making. In *9th International ACM Sigsoft Conference on Quality of Software Architectures (QoSA)*, pages 153–162, 2013.

Sierra, G., Shihab, E., and Kamei, Y. A survey of self-admitted technical debt. *Journal of Systems and Software*, 152(1):70–82, 2019a.

Sierra, G., Tahmid, A., Shihab, E., and Tsantalis, N. Is self-admitted technical debt a good indicator of architectural divergences? In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 534–543, 2019b.

Silva, H. and Valente, M. T. What's in a GitHub star? Understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018.

Silva, M., Terra, R., and Valente, M. T. Does technical debt lead to the rejection of pull requests? In *12nd Brazilian Symposium on Information Systems (SBSI)*, pages 1–7, 2016.

Singer, R. *Shape Up: Stop Running in Circles and Ship Work that Matters.* Idependent, 2019. Available at: `https://basecamp.com/shapeup`.

Spencer, D. *Card Sorting: Designing Usable Categories.* Rosenfeld Media, 2009.

Steinmacher, I., Conte, T. U., Treude, C., and Gerosa, M. A. Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering (ICSE)*, pages 273–284, 2016.

Storey, M.-A., Ryall, J., Bull, R. I., Myers, D., and Singer, J. TODO or to Bug: exploring how task annotations play a role in the work practices of software developers. In *30th International Conference on Software Engineering (ICSE)*, pages 251–260, 2008.

Tan, J., Feitosa, D., and Avgeriou, P. Do practitioners intentionally self-fix technical debt and why? In *37th International Conference on Software Maintenance and Evolution (ICSME)*, pages 251–262, 2021.

Valente, M. T. *Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade.* Independent, 2020.

Vidoni, M. Self-admitted technical debt in r packages: An exploratory study. In *18th International Conference on Mining Software Repositories (MSR)*, pages 179–189, 2021.

Wang, J., Zhang, X., and Chen, L. How well do pre-trained contextual language representations recommend labels for github issues? *Knowledge-Based Systems*, 232:107476, 2021.

Wang, X., Liu, J., Li, L., Chen, X., Liu, X., and Wu, H. Detecting and explaining self-admitted technical debts with attention-based neural networks. In *35th International Conference on Automated Software Engineering (ASE)*, page 871–882, 2020.

Wehaibi, S., Shihab, E., and Guerrouj, L. Examining the impact of self-admitted technical debt on software quality. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 179–188, 2016.

Wiese, M., Rachow, P., Riebisch, M., and Schwarze, J. Preventing technical debt with the TAP framework for technical debt aware management. *Information and Software Technology*, 148:106926, 2022.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. *Experimentation in Software Engineering*. Springer, 2012.

Xavier, L., Ferreira, F., Brito, R., and Valente, M. T. Beyond the code: Mining self-admitted technical debt in issue tracker systems. In *17th International Conference on Mining Software Repositories (MSR)*, pages 137–146, 2020.

Xavier, L., Montandon, J. E., Ferreira, F., Brito, R., and Valente, M. T. On the documentation of self-admitted technical debt in issues. *Empirical Software Engineering*, 27 (163):1–34, 2022a.

Xavier, L., Montandon, J. E., and Valente, M. T. Comments or issues: Where to document technical debt? *IEEE Software*, 39(5):84–91, 2022b.

Xiao, T., Wang, D., Mcintosh, S., Hata, H., Kula, R. G., Ishio, T., and Matsumoto, K. Characterizing and mitigating self-admitted technical debt in build systems. *IEEE Transactions on Software Engineering*, 1:1–1, 2021.

Yli-Huumo, J., Maglyas, A., and Smolander, K. How do software development teams manage technical debt? - an empirical study. *Journal of Systems and Software*, 120 (C):195–218, 2016.

Zampetti, F., Noiseux, C., Antoniol, G., Khomh, F., and Penta, M. D. Recommending when design technical debt should be self-admitted. In *33rd International Conference on Software Maintenance and Evolution (ICSME)*, pages 216–226, 2017.

Zampetti, F., Serebrenik, A., and Penta, M. D. Was self-admitted technical debt removal a real removal? An in-depth perspective. In *15th International Conference on Mining Software Repositories (MSR)*, pages 526–536, 2018.

Zampetti, F., Serebrenik, A., and Penta, M. D. Automatically learning patterns for self-admitted technical debt removal. In *27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 355–366, 2020.

Zazworka, N., Shaw, M. A., Shull, F., and Seaman, C. Investigating the impact of design debt on software quality. In *2nd Workshop on Managing Technical Debt (MTD)*, pages 17–23, 2011.

Zazworka, N., Spínola, R. O., Vetro', A., Shull, F., and Seaman, C. A case study on effectively identifying technical debt. In *17th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, pages 42–47, 2013.