Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems

Laerte Xavier
ASERG Group - Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
laertexavier@dcc.ufmg.br

Fabio Ferreira
Center of Informatics - Federal Institute
of the Southeast of Minas Gerais
Barbacena, Brazil
fabio.ferreira@ifsudestemg.edu.br

Rodrigo Brito
ASERG Group - Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
britorodrigo@dcc.ufmg.br

Marco Tulio Valente
ASERG Group - Department of Computer Science
Federal University of Minas Gerais (UFMG)
Belo Horizonte, Brazil
mtov@dcc.ufmg.br

ABSTRACT

Self-admitted technical debt (SATD) is a particular case of Technical Debt (TD) where developers explicitly acknowledge their sub-optimal implementation decisions. Previous studies mine SATD by searching for specific TD-related terms in source code comments. By contrast, in this paper we argue that developers can admit technical debt by other means, e.g., by creating issues in tracking systems and labelling them as referring to TD. We refer to this type of SATD as issue-based SATD or just SATD-I. We study a sample of 286 SATD-I instances collected from five open source projects, including Microsoft Visual Studio and GitLab Community Edition. We show that only 29% of the studied SATD-I instances can be tracked to source code comments. We also show that SATD-I issues take more time to be closed, compared to other issues, although they are not more complex in terms of code churn. Besides, in 45% of the studied issues TD was introduced to ship earlier, and in almost 60% it refers to Design flaws. Finally, we report that most developers pay SATD-I to reduce its costs or interests (66%). Our findings suggest that there is space for designing novel tools to support technical debt management, particularly tools that encourage developers to create and label issues containing TD concerns.

CCS CONCEPTS

• **Software and its engineering → Software evolution**.

ACM Reference Format:
Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 2020. Beyond the Code: Mining Self-Admitted Technical Debt in Issue Tracker Systems. In *17th International Conference on Mining Software Repositories (MSR '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3379597.3387459

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
MSR '20, October 5–6, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7517-7/20/05…$15.00
https://doi.org/10.1145/3379597.3387459

1 INTRODUCTION

In software development, the "done is better than perfect" maxim reflects the inevitable trade-off between keeping software quality and releasing on time. In this context, the Technical Debt (TD) metaphor—first framed by Cunningham in 1992 [5]—refers to the unavoidable maintenance and evolution costs of such *not-quite-right* solutions. Several circumstances drive developers to assume these debts, such as deadline pressure, existing low quality code, and poor software process [16]. In fact, the term has been widely adopted since its definition [14] and became subject of various studies, mostly regarding its identification [1, 17, 23, 34], management [7, 24, 30], and impact [3, 13, 27, 31, 33].

Self-admitted technical debt (SATD) is a particular case of TD where developers explicitly admit their sub-optimal implementation decisions [2, 18, 21, 32]. However, to our knowledge, SATD studies rely only on source comments to identify SATD instances. Particularly, existing studies search for specific TD-related terms in source code comments — such as *fixme*, *TODO*, and *to be fixed*. By contrast, in this paper we argue that developers can acknowledge technical debt out of the source code, by creating issues in tracking systems documenting their sub-optimal implementation decisions. To document the debt, developers label these issues with terms such as *technical debt* or *debt*. An example is presented in Figure 1, in the next page. 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.

We assume there are at least two types of SATD: (1) **SATD documented using source code comments (SATD-C)**, which has been extensively studied in the past; and (2) **SATD documented using issues (STAD-I)**, which we propose to study in this paper. Moreover, we focus our study on the payment of SATD-I, i.e., only on issues documenting TD that was successfully closed by developers, indicating the admitted TD problem was solved. With this decision, our intention is to study SATD-I instances that had a practical and positive impact on the projects. It also allowed us to investigate the overlap between SATD-I and SATD-C.

We collect and characterize 286 SATD-I instances from five relevant open-source systems, including GitLab (a git-hosting platform that is publicly developed and maintained using its own services),

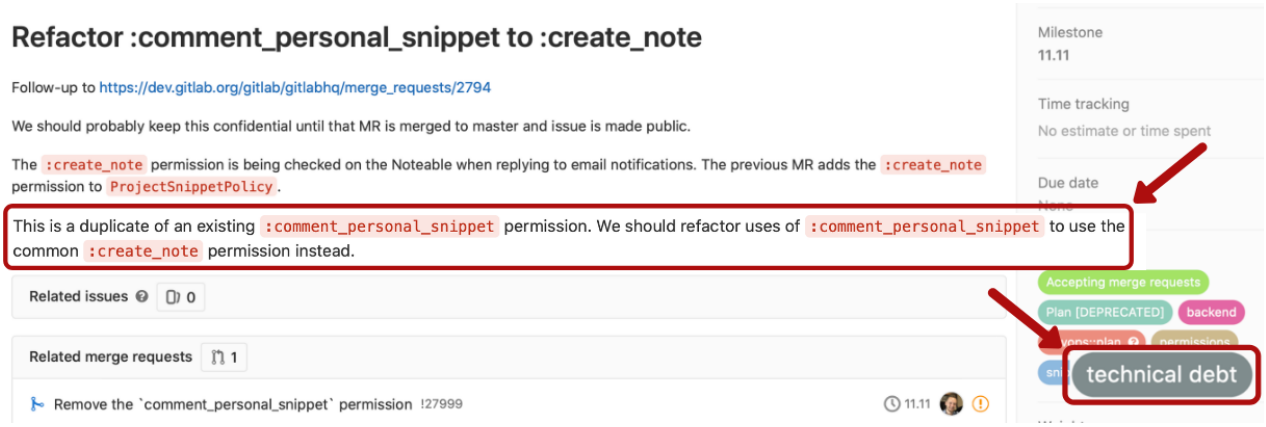


Figure 1: Example of SATD in a GitLab's issue

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 four research questions:

RQ1. What is the overlap between SATD-C and SATD-I? Our intention is to check whether SATD-C and SATD-I refer to different cases of TD. To this purpose, we check whether SATD-I instances are also documented in the source code, i.e., whether there is a correspondence from SATD-I instances to SATD-C found in the source code of the studied systems. To identify this correspondence, we relied on a state-of-the-art tool to detect SATD-C.

RQ2. 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 [15].

RQ3. 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.

RQ4. 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.

We make three key contributions in this paper:

- We identify and study SATD beyond the source code, i.e., detected 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. Finally, we show that there is an overlap between SATD-I and SATD-C but it is not dominant. Only 29% of the studied SATD-I instances can be traced to SATD-C instances.
- We show that SATD-I instances take more time to be closed, compared to other issues, although they are not more complex in terms of code churn. Besides, TD was deliberately introduced in 45% of the studied SATD-I instances, and in almost 60% of them it refers to Design flaws. We also found that most developers paid SATD-I to reduce its interests (66%), and to have a clean code (28%).
- Our findings suggest that there is space for designing and implementing novel tools to support technical debt management, particularly tools that encourage developers to create and label issues containing TD concerns, i.e., to self-admit TD using issues. Moreover, our findings reinforce the importance of introducing TD payment activities as part of software development processes, as a strategy to preserve internal quality or to introduce newcomers to the codebase.

Structure of the paper. In Section 2 we detail our definition of SATD-I. Section 3 presents our dataset of SATD-I instances. In Sections 4, 5 and 6, we answer the proposed research questions. Section 7 provides the main implications of our findings. In Sections 8 and 9, we describe threats to validity and related work, respectively. Finally, Section 10 concludes our work.

2 DEFINITION

We define SATD-I as technical debt instances documented using issues. It contrasts with SATD-C, which is documented using source code comments. Particularly, this definition does not require the developer who introduced the debt in the source to be the same developer responsible for creating the issue in the tracking system. A similar rule is followed by studies targeting SATD-C, which do not check whether the TD-related code and the TD-related comments were inserted by the same author [18, 21, 32]. Therefore, SATD should be viewed as TD acknowledged and detected by the own developers of a software project, without the help of any tool. SATD contrasts, for example, with TD automatically detected using static analysis tools [7, 34].

3 DATASET

In this paper, we study 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 this study (Section 3.1). 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). Finally, we provide a quantitative overview of the proposed dataset (Section 3.3).

3.1 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 last *six months* (December 15th, 2018 to June 15th, 2019). We only selected closed issues because our focus is on 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 we performed with GitLab's developers—and also to increase the confidence on the survey answers (see Section 6).

After applying the described selection criteria, we found 188 issues. The first author of this paper 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, 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 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 [4, 26]. We used GitHub's REST API to search for all issues of such repositories that were closed in the period of December 15th, 2018 to June 15th, 2019—due to the same reasons explained for GitLab—and that include one of the following labels: *technical debt*, *Technical Debt*, and *debt*. 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 for GitLab issues, the first author of this paper inspected all 218 initially selected issues (*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.

[1] https://gitlab.com/gitlab-org/gitlab-ce/issues/34659
[2] https://gitlab.com/gitlab-org/gitlab-ce/issues/44726

3.3 Dataset Characterization

Table I shows the name of the systems in our dataset, 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 now publicly available on GitHub. The remaining SATD-I instances come from influxdata/influxdb (7.3%), mirumee/saleor (3.5%), and nextcloud/server (2.1%). influxdata/influxdb is a framework for time series manipulation and visualization. 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 1: Selected repositories

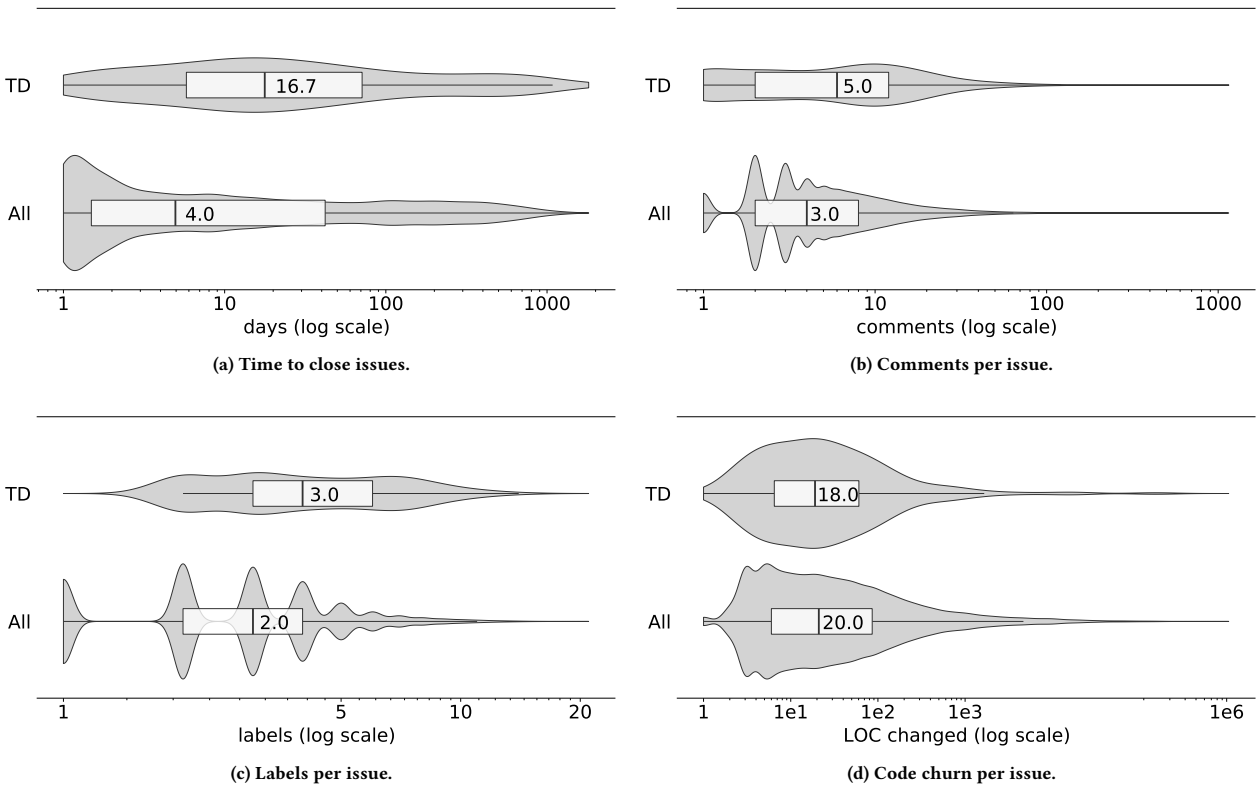
Repository	Tag	SATD-I	%
microsoft/vscode	debt	132	46.2%
gitlab/gilab-ce	technical debt	117	40.9%
influxdata/influxdb	Technical Debt	21	7.3%
mirumee/saleor	technical debt	10	3.5%
nextcloud/server	technical debt	6	2.1%
Total		286	100%

Figure 2 (in the next page) shows violin plots comparing the issues selected in the study with all other issues, i.e., the whole set of issues from the five studied systems. 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 (Figure 2d) shows the code churn of SATD-I versus all issues in our dataset. 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.

> Issues that pay technical debt trigger more discussions and take more time to be closed. However, paying SATD-I does not require larger code churns.

4 OVERLAP BETWEEN SATD TYPES

Self-admitted technical debt (SATD) refers to intentionally implemented Technical Debt, documented by developers either through code comments (SATD-C) or labelled issues (SATD-I), as we proposed in this paper. However, it is not clear whether developers adopt only one of these approaches to admit their debts or there is an overlap between them. In this section, we answer RQ1 by checking if SATD-I instances can be also classified as SATD-C.



(a) Time to close issues.

(b) Comments per issue.

(c) Labels per issue.

(d) Code churn per issue.

Figure 2: Distribution of days, comments, labels, and code churn per issue.

4.1 Methodology

In order to analyze the 286 SATD-I instances to verify whether there are code comments admitting the same debt, we inspected the pull/merge-requests responsible for closing such issues (in GitLab, merge-requests are equivalent to pull-requests). Specifically, we first collected the pull/merge-requests related to the issues (*i.e.,* the ones used to close the issue), retrieving the *diff* of each associated commit. Among the 286 analyzed issues, we found that 167 issues (58.4%) were closed by an associated pull/merge-request. The remaining issues were closed by internal contributors who directly committed the changes, for example. Therefore, in this RQ we focus on these 167 issues, since in this case it is straightforward to navigate from the issues to the pull/merge-requests and then to the commits with the code that pays the SATD-I.

After this data collection phase, we analyzed all the *deleted* lines from the commits paying the TD, in order to investigate whether there were comments indicating SATD-C. The rationale is that such comments should have been removed once the commits are responsible for closing the issues (thus, paying the debt). We used SATDDetector [17] to automatically identify SATD-related comments in the deleted lines. This tool applies NLP techniques to classify textual comments based on a pre-trained model. It is based on a two-phase architecture, responsible for (i) building the model, and (ii) performing the classification. To build the models, the tool's authors relied on a dataset with manually classified source code comments retrieved from eight open source projects [19]. Finally, to perform the classification the tool applies a Naive Bayes Multinomial (NBM) technique.

4.2 Results

SATDDetector analyzed more than 4.7K *diff* chunks associated to the 167 SATD-I instances. The tool identified code comments indicating technical debt admission in 48 issues (28.7%). Table 2 details the results. As we can see, the self-admitted comments are concentrated in two repositories: MICROSOFT/VSCODE, with 26 TD-related issues where the TD was also self-admitted in the code (33.3%); and GITLAB/GILAB-CE, with 22 issues (21.7%). For the remaining repositories, no pull/merge-request was found to be analyzed. Next, we provide examples for both repositories. To facilitate the identification and discussion of the SATD-I instances in this paper, we label them using the initials of the repository name (*i.e.,* VS refers to MICROSOFT/VSCDE; GL to GITLAB/GILAB-CE; IF to INFLUXDATA/INFLUXDB; SL to MIRUMEE/SALEOR; and NX to NEXTCLOUD/SER-VER). The initials are then followed by an integer ID (*e.g.,* GL43 refers to issue 43 from GitLab).

MICROSOFT/VSCODE. In the 26 occurrences of code-based self-admitted technical debt (33.3%), developers include code comments to indicate temporary shortcuts or to guide contributors to fix the implemented solution. As follows:

Table 2: Overlap between SATD-I and SATD-C

Repository	SATD-I	SATD-C	%
microsoft/vscode	78	26	33.3%
gitlab/gilab-ce	89	22	21.7%
influxdata/influxdb	0	-	-
mirumee/saleor	0	-	-
nextcloud/server	0	-	-
Total	167	48	28.7%

HACK: This can be removed once this is fixed upstream xtermjs/xterm.js#1908 (VS122)

If you want to provide a fix or improvement, please create a pull request against the original repository. (VS27)

gitlab/gilab-ce. In 22 SATD-C instances (21.7%), GitLab developers also indicate technical debt by adding code comments, as observed in these issues:

TODO: remove eventHub hack after code splitting refactor (GL90)

Fixes or improvements to automated QA scenarios (GL48)

So you need to move all your global projects under groups or users manually before update or they will be automatically moved to the project owner namespace during the update. When a project is moved all its members will receive an email with instructions how to update their git remote URL. Please make sure you disable sending email when you do a test of the upgrade. (GL84)

> Only 29% of the issues that pay TD can be traced to SATD-C. In other words, 71% of the studied issues document and pay TD that would not be possible to identify by considering only source code documentation.

5 SATD-I CLASSIFICATION

In this section, we present the results of our second research question, regarding the classification of the studied SATD-I. We dedicate Section 5.1 to present the methodology applied to analyze the dataset. Next, we present the classification results as follows: first, in Section 5.2 we discuss issues related to Design (the most popular type of TD) and its corresponding subclassification. In Section 5.3 we present the results for the other types of SATD-I instances.

5.1 Methodology

To identify the types of technical debt paid by developers, we carefully analyzed 286 SATD-I instances using *closed-card sort* [28], 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 authors, (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.* [15]. In this work, the authors describe a systematic mapping to identify and analyze scientific papers on TD from 1992 to 2013. We classify the issues in our dataset according to the following categories proposed by Li *et al.*:

- 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 the build system, 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.

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

- Complex Code: refers to intra-method poorly implemented code.
- 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, three authors of this paper 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 authors. In 178 cases (62.2%) they agreed in the first proposed classification. For the Design subclassification (169 issues), the authors agreed in 92 cases (54.4%). In a final step, the authors discussed each conflict and reached a consensual classification. We use the same identification of Section 4 to label the following discussion.

5.2 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. 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: 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 represents 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)

5.3 Other Types of SATD-I

Table 4 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.

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)

Table 4: 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%

Tests. In 25 cases (8.7%), SATD-related issues report the absence of tests or request improvements on existing tests. 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.

6 SURVEY WITH DEVELOPERS

To answer the remaining research questions, we perform a survey with the developers responsible for closing the 286 SATD-I instances studied in this paper. In this section, we first present the methodology followed in this survey (Section 6.1). After that, we present the results for each research question (Sections 6.2 and 6.3).

6.1 Methodology

We conduct a survey to reveal the reasons why developers introduce technical debt in their code, the motivations of SATD-I payment, and the maintenance problems associated to TD. For that, we sent emails to developers that closed the SATD issues studied in this paper. Specifically, we selected from our 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. 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 between July 11th and August 14th, 2019 (which represents an interval of at most

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: Email sent to developers who paid SATD-I.

six months after the date the issues were closed). Figure 3 shows the template of the survey 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 Martin Fowler [11]. 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 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.

Table 5: Survey answers

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

To interpret the survey answers (1) and (2), the first author followed an *open card-sorting* [28]. 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 discussed the issue. In a final step, the last author reviewed and confirmed the proposed themes. In the following discussion, we label the quotes with D1 to D29 to indicate developers answers.

6.2 Why do developers introduce SATD-I?

To answer this question, we provided two predefined options: the first is related to developers 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 4 presents the obtained results. As we can observe, most of the studied debts were introduced by developers to ship earlier (12 answers). In nine cases, developers were not aware of the TD when it was introduced. Finally, six developers provided other motivations. Next, we detail each of these reasons and provide quotes from extra comments discussed by developers.

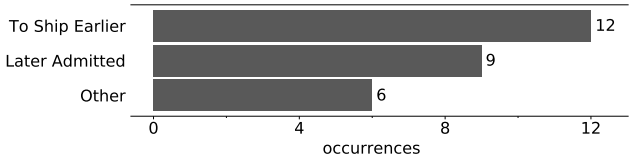


Figure 4: Reasons for introducing SATD-I.

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 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)

SATD admitted after introduction. 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%).

6.3 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 6.1). 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 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 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 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 occurs with the same frequency (six answers for each).

Table 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.

7 IMPLICATIONS

This section presents the study implications on tool support and process improvement.

Tool Support. In RQ1, we show that only 29% of the SATD-I studied instances are also admitted through code comments. Instead, developers tend to create issues reporting debts, labelling them with TD-related tags. Besides, in RQ3 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 paper). 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.,* [12, 17, 20]), 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 RQ4), 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: first, we suggest the formalization of technical debt payment as an actual activity on modern development processes. In agile processes these activities could be introduced as *slacks* to keep developers productive, for example. Second, we advocate that paying technical debt may be included on onboarding activities for new team members [29]. In this case, team leaders would *"delegate this work to newcomers to give them easy stuff to familiarize themselves with the work process"* (D19).

8 THREATS TO VALIDITY

External Validity. This study is 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, the results discussed in Section 6 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, with valuable insights from developers actually responsible for paying SATD-I.

Internal Validity. First, we selected the studied issues by using TD-related labels as a proxy for technical debt identification. However, as discussed by Kruchten *et al.* [14], the concept of technical debt has been diluted since its original proposition. Thus, the misunderstanding of this concept by those who classified the TD-labeled issues would affect the results of our study. To alleviate this threat, the first author of this paper carefully analyzed the initial dataset of 406 TD-labeled issues, and discarded 120 issues (29.6%) that did not have a clear indication of TD payment. Second, we should mention the subjective nature of both closed- and open-card sort used in Sections 5 and 6. Despite the rigor followed by the authors to perform these classifications, 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. Third, 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. Finally, the classification of SATD-C in RQ1 relies on the implementation of a third-party tool proposed in previous studies. Although the tool is vastly used to this end and represent the state-of-the-art to detect SATD-C, the possibility of false positives in the classification may also be reported as a threat.

Construct Validity. This threat concerns the selection of the TD-related issues on GitHub-based projects. As discussed in Section 3, besides GitLab's issues (that we had previous knowledge about its labeling practices), in order to search for analogous issues on GitHub, we mined repositories that included "technical debt", "Technical Debt", and "debt" as labels. Therefore, it is possible that other tags are used to denote TD-related issues.

9 RELATED WORK

Recent research on technical debt derive from the concept of Self-Admitted TD (SATD) presented by Potdar and Shihab [21]. In this work, the authors observe that developers commonly document TD through source code comments. Through the analysis of more than 100K code comments, they find that (i) 2.4%–31% of source code files contain self-admitted technical debt, (ii) experienced developers tend to introduce more debts, and (iii) 26%–63% of SATD gets removed. Bavota and Russo [2] replicate this study on a larger dataset that includes 600K commits and 2 billion comments. The authors first confirm the previous findings, observing that the amount of SATD increases over time and tend to survive a long time in the system. In this paper, we extend their findings by observing that developers may acknowledge TD out of source code. In this case, we define traditional code-base SATD as SATD-C, and investigate the occurrence of issue-based SATD, i.e., SATD-I.

Maldonado *et al.* [18] investigate five Java open source projects with the purpose of examining the amount of TD removed, who performs the removal, how long it lives in a project, and what activities lead to the 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 and the addition of new features. Zampetti *et al.* [32] perform a follow-up study, based on the dataset elicited by Maldonado *et al.* [18], 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. Although these studies also investigate TD removal, we claim that their findings are restricted to SATD-C instances.

Sierra *et al.* [25] 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. Farias *et al.* [9], Huang *et al.* [12] and Liu *et al.* [17] also identify SATD by mining source code comments. Moreover, other studies propose the use of natural language processing (NLP) techniques to support SATD identification [19]. For example, Flisar and Podgorelec [10], Huang *et al.* [12], Ren *et al.* [22], and Fahid *et al.* [8] use machine learning techniques for automating SATD detection. Dai and Kruchten *et al.* [6] improves these approaches by identifying non-code-level technical debt. Maldonado *et al.* [20] proposed a technique to precisely identify SATD, outperforming the current state-of-the-art, based on fixed keywords and phrases. The proposed technique achieved good accuracy for between 80%–90% of the cases. We take advantage of SATDDetector to identify SATD-C in RQ1.

10 CONCLUSION

In this paper, we performed three complementary studies with the purpose of answering four research questions on Self-Admitted Technical Debt documented through labelled issues (SATD-I). We

analyzed 286 SATD-I to (1) investigate the overlap between code-based SATD (SATD-C) and issue-based SATD (SATD-I), (2) identify the types of SATD-I more frequently paid, (3) understand the intentions behind SATD-I insertion, and (4) investigate the reasons why developers pay SATD-I. As a result, we showed that SATD-I instances take more time to be closed, although they are not more complex in terms of code churn. We also revealed that only 29% of the studied SATD-I instances can be tracked to source code comments, and that 45% of the studied debt was introduced to ship earlier. 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). Moreover, our results indicated that most developers paid SATD-I to reduce its interest, and to have a clean code. As practical implications of our studies, we suggest novel tools to support technical debt management. We also discuss how SATD-I payment activities can be introduced as part of software development processes.

As future work, we first intend to enlarge our dataset of SATD-I by mining other tags that may denote TD-related issues. After that, we envision an in-depth analysis of the code transformations performed to pay these debts. Based on this dataset of transformations, we may develop tools and techniques to guide developers on TD payment (*e.g.,* by recommending how to perform changes that contribute to the actual removal of the debt). As a final note, our data is publicly available at: http://doi.org/10.5281/zenodo.3701471.

ACKNOWLEDGMENTS

We thank the 30 developers who participated in our survey and shared their ideas and practices about technical debt payment. This research is supported by grants from FAPEMIG, CNPq, and CAPES.

REFERENCES

[1] Nicolli S.R. Alves, Thiago S. Mendes, Manoel G. de Mendonça, Rodrigo O. Spínola, Forrest Shull, and Carolyn Seaman. 2016. Identification and Management of Technical Debt. *Information and Software Technology* 70, C (2016), 100–121.

[2] Gabriele Bavota and Barbara Russo. 2016. A Large-Scale Empirical Study on Self-Admitted Technical Debt. In *13th Working Conference on Mining Software Repositories (MSR)*. 315–326.

[3] Stephany Bellomo, Robert L. Nord, Ipek Ozkaya, and Mary Popeck. 2016. Got Technical Debt? Surfacing Elusive Technical Debt in Issue Trackers. In *13th International Conference on Mining Software Repositories (MSR)*. 327–338.

[4] Hudson Borges, Andre Hora, and Marco Tulio Valente. 2016. Understanding the Factors that Impact the Popularity of GitHub Repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 334–344.

[5] Ward Cunningham. 1992. The WyCash Portfolio Management System. In *7th Object-oriented Programming Systems, Languages, and Applications (OOPSLA)*. 29–30.

[6] Ke Dai and Philippe Kruchten. 2017. Detecting Technical Debt through Issue Trackers. In *5th International Workshop on Quantitative Approaches to Software Quality (QuASoQ)*. 59–65.

[7] Neil A. Ernst, Stephany Bellomo, Ipek Ozkaya, Robert L. Nord, and Ian Gorton. 2015. Measure It? Manage It? Ignore It? Software Practitioners and Technical Debt. In *10th Joint Meeting on Foundations of Software Engineering (FSE)*. 50–60.

[8] Fahmid M. Fahid, Zhe Yu, and Tim Menzies. 2019. Better Technical Debt Detection via SURVEYing. *arXiv preprint arXiv:1905.08297* (2019).

[9] MÃąrio AndrÃĪ Farias, JosÃĪ AmÃćncio Santos, Marcos Kalinowski, Manoel MendonÃğa, and Rodrigo Oliveira Spinola. 2016. Investigating the Identification of Technical Debt through Code Comment Analysis. In *18th International Conference on Enterprise Information Systems (ICEIS)*. 284–309.

[10] Jernej Flisar and Vili Podgorelec. 2019. Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. *IEEE Access* 7, 1 (2019), 106475–106494.

[11] Martin Fowler. [n.d.]. TechnicalDebtQuadrant. https://martinfowler.com/bliki/TechnicalDebtQuadrant. [accessed 10-October-2019].

[12] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.

[13] Yasutaka Kamei, Everton Da Silva Maldonado, Emad Shihab, and Naoyasu Ubayashi. 2016. Using analytics to quantify the interest of self-admitted technical debt. In *1st International Workshop on Technical Debt Analytics (TDA)*. 68–71.

[14] Philippe Kruchten, Robert L. Nord, and Ipek Ozkaya. 2012. Technical Debt: from Metaphor to Theory and Practice. *IEEE Software* 29, 6 (2012), 18–21.

[15] Zengyang Li, Paris Avgeriou, and Peng Liang. 2015. A Systematic Mapping Study on Technical Debt and Its Management. *Journal of Systems and Software (JSS)* 101, C (2015), 193–220.

[16] Erin Lim, Nitin Taksande, and Carolyn Seaman. 2012. A Balancing Act: what Software Practitioners Have to Say About Technical Debt. *IEEE Software* 29, 6 (2012), 22–27.

[17] Zhongxin Liu, Qiao Huang, Xin Xia, Emad Shihab, David Lo, and Shanping Li. 2018. SATD Detector: A Text-Mining-Based Self-Admitted Technical Debt Detection Tool. In *40th International Conference on Software Engineering: Companion (ICSE-Companion)*. 9–12.

[18] Everton Da Silva Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. 2017. An Empirical Study on the Removal of Self-Admitted Technical Debt. In *33rd International Conference on Software Maintenance and Evolution (ICSME)*. 238–248.

[19] Everton Da Silva Maldonado and Emad Shihab. 2015. Detecting and quantifying different types of self-admitted technical Debt. In *7th International Workshop on Managing Technical Debt (MTD)*. 9–15.

[20] Everton Da Silva Maldonado, Emad Shihab, and Nikolaos Tsantalis. 2017. Using Natural Language Processing to Automatically Detect Self-Admitted Technical Debt. *IEEE Transactions on Software Engineering* 43, 11 (2017), 1044–1062.

[21] Aniket Potdar and Emad Shihab. 2014. An Exploratory Study on Self-Admitted Technical Debt. In *30th International Conference on Software Maintenance and Evolution (ICSME)*. 91–100.

[22] Xiaoxue Ren, Zhenchang Xing, Xin Xia, David Lo, Xinyu Wang, and John Grundy. 2019. Neural Network-based Detection of Self-Admitted Technical Debt: from Performance to Explainability. *ACM Transactions on Software Engineering and Methodology* 28, 3 (2019), 1–45.

[23] Nicolli Rios, Manoel Gomes de MendonÃğa Neto, and Rodrigo Oliveira SpÃŋnola. 2018. A tertiary study on technical debt: types, management strategies, research trends, and base information for practitioners. *Information and Software Technology* 102, 1 (2018), 117 – 145.

[24] Giancarlo Sierra, Emad Shihab, and Yasutaka Kamei. 2019. A survey of self-admitted technical debt. *Journal of Systems and Software* 152, 1 (2019), 70 – 82.

[25] Giancarlo Sierra, Ahmad Tahmid, Emad Shihab, and Nikolaos Tsantalis. 2019. Is Self-Admitted Technical Debt a Good Indicator of Architectural Divergences?. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 534–543.

[26] Hudson Silva and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129.

[27] Marcelino Silva, Ricardo Terra, and Marco Tulio Valente. 2016. Does Technical Debt Lead to the Rejection of Pull Requests?. In *12nd Brazilian Symposium on Information Systems (SBSI)*. 1–7.

[28] Donna Spencer. 2009. *Card Sorting: Designing Usable Categories*. Rosenfeld Media.

[29] Igor Steinmacher, Tayana Uchoa Conte, Christoph Treude, and Marco Aurélio Gerosa. 2016. Overcoming Open Source Project Entry Barriers with a Portal for Newcomers. In *38th International Conference on Software Engineering (ICSE)*. 273–284.

[30] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. 2008. 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)*. 251–260.

[31] Sultan Wehaibi, Emad Shihab, and Latifa Guerrouj. 2016. Examining the Impact of Self-Admitted Technical Debt on Software Quality. In *23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 179–188.

[32] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 2018. Was Self-Admitted Technical Debt Removal a Real Removal? An In-Depth Perspective. In *15th International Conference on Mining Software Repositories (MSR)*. 526–536.

[33] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. 2011. Investigating the Impact of Design Debt on Software Quality. In *2nd Workshop on Managing Technical Debt (MTD)*. 17–23.

[34] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. 2013. A Case Study on Effectively Identifying Technical Debt. In *17th International Conference on Evaluation and Assessment in Software Engineering (EASE)*. 42–47.