Comments or Issues: Where to Document Technical Debt?

Laerte Xavier, João Eduardo Montandon, Marco Tulio Valente

Federal University of Minas Gerais, Brazil
{laertexavier,joao.montandon,mtov}@dcc.ufmg.br

Abstract. Self-Admitted Technical Debt (SATD) is a form of Technical Debt where developers document the debt using source code comments (SATD-C) or issues (SATD-I). However, it is still unclear the circumstances that drive developers to choose one or another. In this paper, we survey authors of both types of debts using a large-scale dataset containing 74K SATD-C and 20K SATD-I instances, extracted from 190 GitHub projects. As a result, we provide 13 guidelines to support developers to decide when to use comments or issues to report Technical Debt.

Keywords. Technical Debt; Self-Admitted Technical Debt; Documentation.

1 Introduction

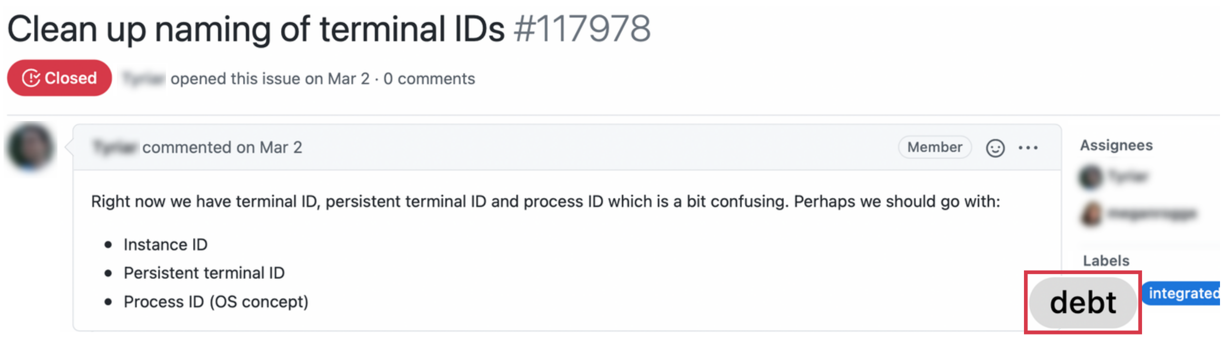
Modern software developers are under constant pressure to evolve their systems in order to preserve existing clients or to explore new markets. During this process, it is inevitable to incur in sub-optimal technical decisions, whose accumulation results in what is called *Technical Debt* (TD) [3]. However, developers also know that TD eventually emerges in the form of features that are more risky and difficult to implement. Therefore, it is not a surprise to observe developers documenting TD instances, which the literature refers to *Self-Admitted Technical Debt* (SATD) [6].

Previous studies on SATD focused mostly on using source code comments to this purpose [1, 5, 7, 10]. For example, 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 or improved in future sprints. As an example, we have the following comment extracted from PYTORCH/PYTORCH:

```
# TODO If this codepath becomes popular, it may be worth taking a look at optimizing
```

```
it -- for now a simple implementation is used.
```

However, in many projects developers increasingly rely on issue tracking systems as a medium to discuss among themselves several aspects of software development, including technical debt. Particularly, they can document TD by opening issues in tracking systems, as illustrated in Figure 1. In this case, TD is indicated by labels such as *technical debt*, *debt*, and *workaround*.


Clean up naming of terminal IDs #117978

Closed [redacted] opened this issue on Mar 2 · 0 comments

[redacted] 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)

Assignees
[redacted]

Labels
debt
integrated

Figure 1: Example of SATD in MICROSOFT/VSCODE issues.

Although in less number, recent papers study SATD as documented in issues. For example, in a previous work [9], we studied 286 TD-related issues from five well-known projects, including GitLab and Microsoft's VS Code. We confirmed that developers rely on issues to document and discuss TD. As a second example, Li *et al.* [4] manually investigated a sample of 500 issues from two open source projects (Hadoop and Camel). In 117 issues, they found discussions about TD.

However, **there is still a lack of knowledge on the circumstances that drive developers to choose between code comments (SATD-C) and issues (SATD-I) to document TD**. To tackle this question, in this article, we first build a dataset of 20,265 SATD-I instances and 74,306 SATD-C instances, extracted from 190 GitHub projects. We use this dataset to conduct a survey with developers who documented TD using comments and issues, as they have practical experience with both forms of SATD. We then report the factors they consider to choose between SATD-C and SATD-I.

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 mined SATD-C and SATD-I instances in the top-5K most starred GitHub repositories. To select SATD-I instances, we used TD-related labels as proxy to identify issues reporting TD [9]. We analyzed 97,106 labels and applied the following cleaning steps:

1. We discarded 60,032 labels associated with less than 10 issues.
2. We removed 19,209 labels by adopting multiple regular expressions with well-known labels that *do not* denote TD [2] (*e.g., bug*, *enhancement*, *feature*).
3. The first author manually analyzed the remaining 17,865 labels in order to select labels explicitly denoting technical debt. *e.g., tech debt*, *debt*, *cleanup*, *workaround*. The second author validated this selection by analyzing 500 randomly selected labels.

As a result, we identified 219 TD-related labels, associated with 190 repositories in our top-5K initial selection (*i.e.,* we traced back the labels to their corresponding repositories). Then, we used GitHub's API to retrieve the issues marked with these labels. As a result, we found 20,265 SATD-I instances.

To select SATD-C instances, we statically analyzed the source code of the same 190 repositories. For each file, we selected code comments containing one of the following terms: *TODO*, *workaround*, *fixme*, and *hack*. We restricted our analysis to these keywords as a previous study listed them among the most frequent SATD-C patterns [6]. We retrieved 74,306 comments, distributed through 182 repositories. To validate this selection, the first author inspected 3K comments (randomly selected). He confirmed all of them indeed refer to SATD-C. However, in order to have a second opinion, the second author 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. Therefore, we did not find any false positives, certainly because our keywords (*TODO*, *workaround*, *fixme*, and *hack*) when used in comments are clear and widely-established indicators of SATD.

Based on the 20K SATD-I and 74K SATD-C retrieved instances, we leveraged the list of developers who created at least one of these debts: the author who opened the issue for SATD-I, and the commit author for SATD-C. We identified 8,082 authors. Figure 2 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.

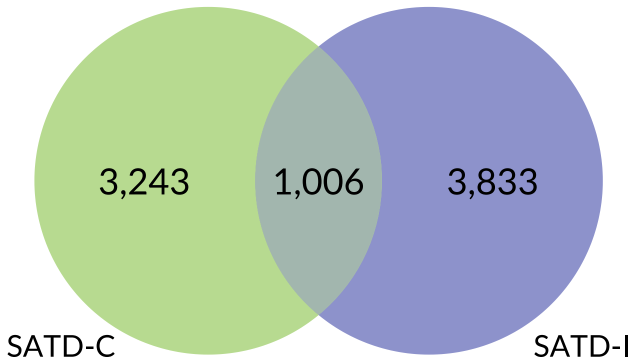


Figure 2: Number of developers who reported each type of SATD.

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. Our survey was exempted from approval by our university's Institutional Review Board (IRB) because it only involves non-Brazilian participants.

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 first author followed an open-card sorting [8] 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 first author 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 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, the second author 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.

3 Proposed Guidelines

As presented in Table 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.

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

Table 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

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)

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:

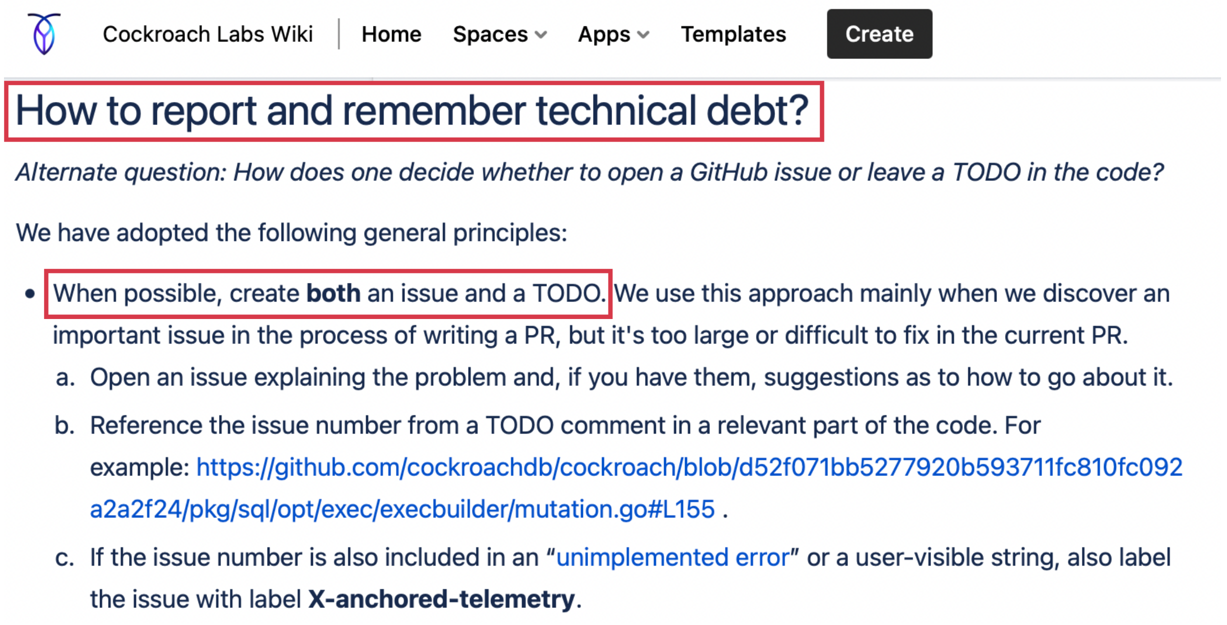
GitHub issues are particularly useful in the [project] because they can be tagged with '/good-first-issue' which enables new contributors to find things to work on and get familiar with the project. (D10)

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 2). Figure 3 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 3: Guidelines included in COCKROACHDB/COCKROACH wiki, after the discussion raised by our survey.

4 Conclusion

In this paper, 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 74,306 SATD-C instances. We used the obtained answers to unveil practical guidelines to support developers to better document TD.

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 4 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 3). 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.

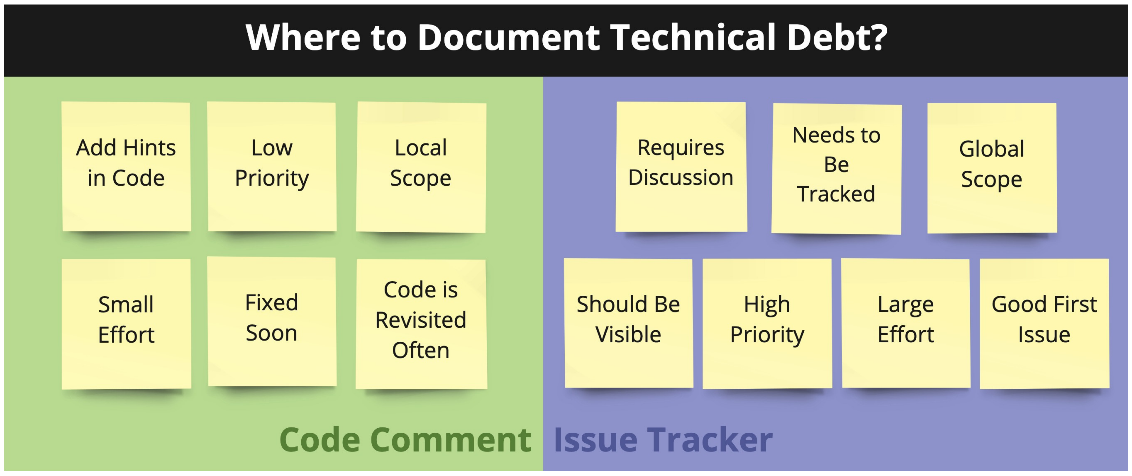


Figure 4: Technical Debt documentation guidelines.

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 study is publicly available at `https://doi.org/10.5281/zenodo.6418088`.

Acknowledgments

Our research is supported by CNPq, FAPEMIG and CAPES.

References

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

[2] J. Cabot, J. L. Cánovas Izquierdo, V. Cosentino, and B. Rolandi. 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.

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

[4] Yikun Li, Ohamed Soliman, and Paris Avgeriou. 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.

[5] Everton Da Silva Maldonado, Rabe Abdalkareem, Emad Shihab, and Alexander Serebrenik. An empirical study on the removal of self-admitted technical debt. In *33rd International Conference on Software Maintenance and Evolution (ICSME)*, pages 238–248, 2017.

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

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

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

[9] Laerte Xavier, Fabio Ferreira, Rodrigo Brito, and Marco Tulio Valente. 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.

[10] Fiorella Zampetti, Alexander Serebrenik, and Massimiliano Di Penta. 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.

About the authors

Laerte Xavier is a PhD student in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he is a member of the Applied Software Engineering Research Group (ASERG). His research interests include software architecture, software maintenance and evolution, and software quality analysis. Laerte received a Master's Degree in Computer Science from the Federal University of Minas Gerais. Contact him at `laertexavier@dcc.ufmg.br`.

João Eduardo Montandon is assistant professor at the Colégio Técnico (COLTEC) of Federal University of Minas Gerais (UFMG) since 2014. He is currently a member of the Applied Software Engineering Research Group (ASERG), where he conducts research focused on empirical software engineering, mining software repositories, and software maintenance and evolution. He received his Ph.D. degree in Computer Science from the Department of Computer Science of Federal University of Minas Gerais (UFMG). Contact him at `joao.montandon@dcc.ufmg.br`, or visit `https://jem.af`.

Marco Tulio Valente is an associate professor in the Computer Science Department at the Federal University of Minas Gerais (UFMG), where he is also a member of the Applied Software Engineering Research Group (ASERG). His research interests include software architecture, software maintenance and evolution, and software quality analysis. Valente received a Ph.D. in Computer Science from the Federal University of Minas Gerais. Contact him at `mtov@dcc.ufmg.br`, or visit `www.dcc.ufmg.br/~mtov`.