Empirical Software Engineering (2021) 26: 131
https://doi.org/10.1007/s10664-021-10031-3

Self-admitted technical debt practices: a comparison between industry and open-source

Fiorella Zampetti[1] · **Gianmarco Fucci**[1] · **Alexander Serebrenik**[2] · **Massimiliano Di Penta**[1]

Accepted: 4 August 2021 / Published online: 27 September 2021
© The Author(s), under exclusive licence to Springer Science+Business Media, LLC, part of Springer Nature 2021

Abstract

Self-admitted technical debt (SATD) consists of annotations, left by developers as comments in the source code or elsewhere, as a reminder about pieces of software manifesting technical debt (TD), i.e., "not being ready yet". While previous studies have investigated SATD management and its relationship with software quality, there is little understanding of the extent and circumstances to which developers admit TD. This paper reports the results of a study in which we asked developers from industry and open-source about their practices in annotating source code and other artifacts for self-admitting TD. The study consists of two phases. First, we conducted 10 interviews to gather a first understanding of the phenomenon and to prepare a survey questionnaire. Then, we surveyed 52 industrial developers as well as 49 contributors to open-source projects. Results of the study show how the TD annotation practices, as well as the typical content of SATD comments, are very similar between open-source and industry. At the same time, our results highlight how, while open-source code is spread of comments admitting the need for improvements, SATD in industry may be dictated by organizational guidelines but, at the same time, implicitly discouraged by the fear of admitting responsibilities. Results also highlight the need for tools helping developers to achieve a better TD awareness.

Keywords Technical debt · Self-admitted technical debt · Empirical study · Software quality

Communicated by: Lin Tan

✉ Fiorella Zampetti
fiorella.zampetti@unisannio.it

Gianmarco Fucci
gianmarco.fucci@unisannio.it

Alexander Serebrenik
a.serebrenik@tue.nl

Massimiliano Di Penta
dipenta@unisannio.it

[1] University of Sannio, Via Traiano, 3, Benevento, Italy

[2] Eindhoven University of Technology, Eindhoven, The Netherlands

1 Introduction

The notion of Technical Debt (TD) has been introduced by Cunningham (1992) as "not quite right code which we postpone making it right". Awareness has been empirically shown to be a very important factor when managing TD (Ernst et al. 2015), for making managers and, ultimately, end-users, knowledgeable of the effort and activities necessary for software improvement (Lim et al. 2012). Building on and refining the notion of TD, Potdar and Shihab (2014) introduced the concept of Self-Admitted Technical Debt (SATD) as a sub-category of TD capturing "intentional (i.e., self-admitted) quick or temporary fixes, or in general source code that needs to be improved (i.e., technical debt)". Examples of SATD by Potdar and Shihab (2014) are such comments as "// TODO this is such a hack it is silly" from Eclipse and "// Unsafe; should error" from Chromium OS. Following the seminal work of Potdar and Shihab (2014) researchers have studied different kinds of TD (Bavota and Russo 2016; Fucci et al. 2021; Maipradit et al. 2020; Mensah et al. 2018; Rantala et al. 2020): from those introduced by such keywords as "TODO" or "FIXME" to those focusing on situations when developers are waiting for a certain event or an updated functionality, from those focusing on functional defects to those related to maintainability. To encompass different variants of SATD found in the scientific literature, in this paper we opt for a broad definition and consider as SATD any source code comments for annotating delayed or intended work activities such as TODO, FIXME, hack, workaround.

From an observer's perspective, SATD represents the evidence of TD in source code. Previous studies have related such evidence with software quality (Bavota and Russo 2016; Wehaibi et al. 2016; da S Maldonado et al. 2017a; Zampetti et al. 2018), by mining and analyzing related comments in the source code or elsewhere, e.g., in the issues (Xavier et al. 2020). However, while studies have been conducted to understand developers' perception of TD (Ernst et al. 2015), to identify strategies related to its introduction (Fucci et al. 2020) and removal (Bavota and Russo 2016; da S Maldonado et al. 2017a; Zampetti et al. 2018; Iammarino et al. 2019; Zampetti et al. 2020) so far there is limited empirical evidence on the reasons and circumstances in which developers admit TD under the form of a SATD comment, and how they cope with it beyond removal.

An SATD comment in the source code is the result of decisions taken by a developer whether to annotate source code and, when doing so, what details to include in the SATD annotation. Such a decision might be influenced not only by the developers' evaluation of the problematic nature of the code but also by organizational and project culture and guidelines. For example, while 88% of the survey respondents from the Dutch ING bank report SATD (Vassallo et al. 2016), opinions on the usefulness of such annotations vary[1] and as such not every developer might feel compelled to report SATD.

This paper sheds light on the technical debt annotation practices from developers' perspective (both industrial and open-source). We start looking at whether or not developers annotate design decisions at all, going deeper on those decisions that introduce TD in the

[1] See, e.g., the "Todo Comments Considered Harmful" http://wiki.c2.com/?TodoCommentsConsideredHarmful vs. "Todo Comments Considered Useful" http://wiki.c2.com/?TodoCommentsConsideredUseful.

source code. Furthermore, since developers might have different reasons for (not) admitting SATD, e.g., organizational policies, we look at what are the main reasons behind their (lack of) admissions. Moreover, considering that previous work has highlighted that developers might use different channels for annotating TD (Potdar and Shihab 2014; Xavier et al. 2020), we gather a broad knowledge about the channels used for reporting SATD. Other than looking at the channels, it is important to properly understand what is the content of SATD annotations instead of simply considering the artifacts to which the SATD pertains to, e.g., requirements, test or design. Finally, we also look at what are the developers' reaction while modifying a piece of code affected by a SATD, e.g., address it or simply leverage it to influence other design and implementation decisions.

To address the aforementioned goal, we conducted 10 interviews with developers, to gather a first understanding of the studied phenomenon. The provided answers allowed us to design a survey, which was sent to both industrial developers (mostly personal contacts) and mailing lists of open-source systems (OSS), asking them to answer questions related to their SATD annotation practices in those specific contexts. The survey received a total of 101 responses (52 from the industry, and 49 from the open-source). Differently from previous studies analyzing SATD by looking at the source code and its comments (Bavota and Russo 2016; Wehaibi et al. 2016; Zampetti et al. 2018), we are more interested to understand developers' rationale and perception. For this reason, the study has been entirely conducted through interviews and surveys rather than by mining software repositories.

At the first glance, the study results indicate that the SATD practice—at least as reported by the study respondents—is very similar in industry and OSS. Nevertheless, some results highlight several organizational and cultural differences: e.g., industrial developers are less prone to admit TD than those working in OSS, either because of organizational guidelines, or because of a culture of "hiding under-performance", which may reveal counter-productive. Results also highlight the need for better developers' support in admitting TD and make it easily traceable and understandable by others.

By studying the extent to which developers admit TD, whether there are barriers or challenges threatening this admission, and by understanding how SATD is being used by developers, researchers could better scope the development of tools supporting TD management. Also, project managers can learn from this study in terms of benefits derived from admitting TD, and in forming "better practices" concerning TD documentation and management.

The paper is organized as follows. Section 2 provides details about the study design and planning. Results are reported in Section 3, while their implications are discussed in Section 4, and the threats to the study validity in Section 5. After a discussion of related work (Section 6) and Section 7 concludes the paper and outlines directions for future work.

The study material and datasets are available in a replication package (Zampetti et al. 2021).

2 Study Design

The *goal* of this study is to understand the TD annotation practices from developers' perspective. The study *perspective* is of researchers, that want to understand how annotation practices work, and possibly differ, in industry and open source. The *context* of the study consists of 111 developers working either on industrial projects or contributing in OSS, reached through semi-structured interviews and a survey. It is important to note that the

distinction between industry and open source may not be sharp, because industrial developers may contribute to OSS. However, as clarified in Section 5, we limited this threat by clarifying, in the survey and interviews, the context for which respondents should reply to our questions.

We aim at addressing the following five research questions:

RQ$_1$ *To what extent developers admit SATD?* We intend to understand whether developers document design decisions at all (e.g., through comments in the source code), but also when the source code is not ready yet (and, thus, they admit TD). Previously, Vassallo et al. (2016), by surveying 152 developers of a large financial organization, to analyze the interaction between continuous integration and delivery practices and development activities, found that 88% of the respondents admit TD. Differently from their work, we aim at understanding the frequency of such an admission beyond a specific company and from different contexts, i.e., industrial and open-source (OSS).

RQ$_2$ *What are the reasons for (not) admitting SATD?* Developers might have valid reasons for admitting or not admitting SATD. These reasons can be related to a developer's specific behavior or organizational policies.

RQ$_3$ *What are the channels and tool support used to admit TD?* The notion of Self-Admitted Technical Debt has been introduced by Potdar and Shihab (2014) as technical debt admitted by developers by using source code comments. In a recent study, Xavier et al. (2020) conjectured that developers can admit TD by creating issues in tracking systems. We conjecture that developers may rely on other different channels, such as internal mailing list or messaging apps, to annotate design decisions including the ones introducing a TD in the source code. Differently from previous studies on SATD, with this research question, we aim at gathering a broad knowledge about the channels mainly used for admitting TD by explicitly asking developers.

RQ$_4$ *What is the content of SATD annotations?* Previous work (Bavota and Russo 2016) has classified SATD by simply looking at the software artifacts it belongs to, e.g., requirements, test, or design, without considering the content of the admission. Differently from previous studies, our goal is to classify the content of the SATD annotations from the developers' perspective by asking them what TD and improvement intentions they usually admit, e.g., expresses the need for refactoring or performance improvement, indicates source code that works only under certain conditions or does not adequately handle exceptions, or serves as a reminder that further activities should be performed when a next version of an external API is released.

RQ$_5$ *How do developers react when they encounter SATD comments?* While previous research has studied the extent to which SATD is addressed, or in general removed from the source code (da S Maldonado et al. 2017a; Zampetti et al. 2018), to the best of our knowledge there are no studies aimed at understanding how developers act on pieces of code that need to be evolved while being affected by a previously admitted TD. For this reason, our last research question aims to shed light on whether, in general, developers take previously admitted TD into account, e.g., address it before evolving the affected code or leverage it to influence other design and implementation decisions.

2.1 Methodology Overview

The methodology of this study follows two phases. First, we rely on semi-structured interviews to gather a broad knowledge of the topics treated in the five research questions. Then, based on the information acquired from the interviews, we design a survey questionnaire, which complements the interviews and aims at enlarging the sample of answers. This survey is sent to professional developers working in the industry, as well as to mailing lists of 104 open source projects. It is important to highlight that while some of our research questions might be also addressed by looking at SATD comments in the source code (i.e., RQ_1 and RQ_4), we advisedly focused only on the developers' perspective by using both semi-structured interviews and a survey since investigating the reasons behind the (not) admission as well as the developers' reactions while encountering SATD comments in the source code may only be addressed by directly asking them.

In the following, we detail the methodology followed at each phase of the study.

2.2 Interview Design and Methodology

The *context* of the interviews consists of 10 professional developers (referred to as I_n below), working in 7 different software companies, and reached through personal contacts. We have involved developers of small, midsize and large companies, in order to achieve diversity in terms of organizational culture.

First, we collected participants' demographics, i.e., academic background, kind of organization where they work, development experience, programming languages, IDEs, and other tools they used. We have included a question about IDEs since some IDEs support such annotations as FIXME and TODO that might prompt developers to admit technical debt. Then, the interview structure featured questions following the research questions stated above (the detailed structure is in our replication package (Zampetti et al. 2021)). We ask:

1. whether, and how frequently developers insert comments in the source code for annotating delayed or intended work activities, hacks, or workarounds;
2. what are the main reasons for code annotations, and what are the main reasons for refraining from using code annotations;
3. besides source code comments, what are other channels for such annotations, such as chats, code reviews, issue reports, and task management systems;
4. what is the typical content and level of detail of code annotations; and
5. how developers react when they find an annotation in the source code.

The interviews have been conducted by two of the authors, either in person, through a conference call, or (in one case) through chat (as this was the interviewee's preferred communication medium). The interview was either recorded (prior consent), or answers were handwritten on paper.

2.3 Interview Data Analysis

After the interviews were completed, the answers were copied into an online spreadsheet, grouped along the categories described above. Then, one of the author extracted notes from the interviews' transcripts and, relying on an open card sorting process (Spencer 2009; Zimmermann 2016), defined for each question a finite set of possible options (i.e., labeled group of answers) to be used while designing the survey structure. The labeled group of answers

have been validated by two different authors who could propose changes, and after a final discussion with the first author, we derived the set of alternatives to use while structuring the survey.

2.4 Survey Design and Methodology

The survey design closely follows the interview structure, with some key differences, mainly having the goal of reducing the time needed to provide answers and to limit/avoid abandonment. Specifically, our survey includes the questions used to guide our semi-structured interviews, for which we used the finite set of options identified while applying the open-card sorting procedure previously described. In other words, for the survey we opt for multiple-choice questions with answer options derived from the findings of the interviews. For most questions, multiple answers could have been selected. To mitigate the risk of potential incompleteness in the answer options, we provided the respondents with a possibility of including a different answer in the free text space.

Furthermore, based on the results collected from the interviews we added two open-ended questions aimed at gaining insights about the annotation policies adopted for (not) admitting TD. Table 1 shows the structure of the survey in which we have highlighted the linking with the structure used for the semi-structured interviews together with how the questions are used to answer the research questions previously described. Specifically, the

Table 1 Questions provided in the semi-structured interviews (I) and in the survey (S) together with their mapping to the research questions their answers are going to address

ID	Question body	I+S/S	RQ
$\mathbf{Q_1}$	How frequently do you insert comments for annotating your implementation and design choices?	I+S	RQ_1
$\mathbf{Q_2}$	How frequently do you insert comments for annotating delayed or intended work activities?	I+S	RQ_1
$\mathbf{Q_3}$	Does your organization have specific policies regarding the addition of annotations for the implementation and design choices? If yes, please describe them.	S	RQ_2
$\mathbf{Q_4}$	Does your organization have specific policies regarding the addition of annotations for delayed or intended work activities? If yes, please describe them.	S	RQ_2
$\mathbf{Q_5}$	What are the tracing mechanism mainly used for annotating your implementation and design choices?	I+S	RQ_3
$\mathbf{Q_6}$	While annotating your implementation and design choices in the source code do you rely on IDE-supported annotations (e.g., automatically generated TODO)?	I+S	—
$\mathbf{Q_7}$	What are the main motivations for annotating the implementation and design choices?	I+S	RQ_2
$\mathbf{Q_8}$	What are the main motivations for not annotating your implementation and design choices?	I+S	RQ_2
$\mathbf{Q_9}$	What is the typical content that you usually include while annotating delayed or intended work activities?	I+S	RQ_4
$\mathbf{Q_{10}}$	If you find an annotation reporting that the code is not in the right shape, in the code while implementing a new feature or improving an existing feature, what do you actually do?	I+S	RQ_5

third column reports whether we used the questions in both the interviews and the survey (I+S), or only in the survey (S). For the complete structure of the survey you can refer to our replication package (Zampetti et al. 2021).

Finally, the demographics questions were asked at the end of the questionnaire, and made optional. This has been done to reduce the stereotype threat (Steele and Aronson 1995), and also allowed us to obtain responses from individuals that might be reluctant to disclose the demographics information.

The questionnaire has been made available online using Google Forms.[2]

To gather responses from professional developers, we mainly relied on personal contacts, but we also shared the questionnaire on Twitter and Reddit channels about software engineering, software development, and agile practices/SCRUM. Since this is an exploratory study, we prefer to rely on a convenience sampling, as previously done in literature (Arnaoudova et al. 2014; Rastkar et al. 2014; Celik et al. 2016; Di Nucci et al. 2017; Wei et al. 2017). This is because software developers represent a hidden population so we did not have a sampling frame (Baltes and Ralph 2020). By using non-probabilistic sampling helps us to conveniently reach a suitable number of study participants. We received 52 answers, of which most of them (75%) have been reached through our personal contacts.

For what concerns OSS, we started with five Java projects from the dataset of da S Maldonado et al. (2017a). To complement this data, we consider five popular programming languages on GitHub, i.e., C/C++, Javascript, Java, Python, and Ruby. While we conjecture that, in general, SATD practices are programming language-agnostic, we cannot exclude that there could be some language-specific peculiarities when admitting TD, e.g., related to API management, language-specific features, etc. Therefore, to ensure enough generalizability of the results, we targeted projects written in multiple programming languages.

For each programming language, we select the top 100 projects ordered accounting for the number of forks. From such lists, we pick projects (i) having public developers' mailing lists, forum or discussion groups, and (ii) preferring projects with a greater number of contributors. All the considered projects have more than 30 contributors. Once selected, for each project, one of the authors subscribed to the mailing list/forum and posted the questionnaire together with an explanation letter. We kept posting across projects developed with different languages, expanding the list of top projects if necessary, balancing the number of responses achieved for each language, and aiming at an overall number of responses comparable to those of industrial developers. In the end, we shared the survey with five projects from the dataset of da S Maldonado et al. (2017a), as well as with 10 Java, 27 Ruby, 33 C/C++, 15 Javascript, and 14 Python GitHub projects. We obtained 49 answers, of which 11 from Java, 10 from Ruby, 10 from Python, 11 from C/C++, and 7 from Javascript projects.

One important note concerns how to deal with developers working both in industry and in the open-source. When administering the survey questionnaire, since developers could work both in open-source and industry, we explicitly asked them to report their behavior related to their industrial project (when we contacted them as part of the industrial questionnaire) and their behavior in the open-source projects when contacting them through the OSS mailing lists.

[2]No participant, nor any potential participant reported us to have any particular privacy concern about this channel.

2.5 Survey Data Analysis

After having collected all responses, two of the authors performed an open coding of answers to questions where this was required, and specifically for those about policies for annotating design decisions and SATD. Similarly to the procedure described in Section 2.3, the first author extracted codes from the free-text left by our survey respondents and tried to group the codes in more cohesive categories, each one with a specific label. This classification has been validated by a different author who could propose changes that have been discussed together with the first author to reach a consensus.

Results are presented by (i) discussing the interviews' main findings, and (ii) showing the survey outcome in the form of bar charts, as well as discussing specific answers provided by respondents. To support the comparison between results obtained in industry and OSS, we perform statistical analysis as follows. For RQ_1, the survey asked about the annotation frequency for design decisions and SATD, and possible answers are (i) never, (ii) 25% of the development tasks, (iii) between 25 and 75% of the development tasks, and (iv) above 75% of the development tasks. To compare industry and OSS, we encoded the answers into integers, and then used a two-tailed Wilcoxon Rank Sum test (Wilcoxon 1945) and Cliff's delta effect size (Grissom and Kim 2005). The employed test is a non-parametric test for independent samples. We used a non-parametric test because, according to Wilk-Shapiro test, data significantly deviates from normality (p-value< 0.001).

For the other questions, the survey contained multiple-choice questions with answer options and that for many questions respondents have been encouraged to indicate all answers applying to them. Hence, we do not compare different answer options with each other but the proportions of respondents from industry and OSS that have chosen a specific answer option. To this end, we first construct a confusion matrix containing (i) the number of industry respondents who picked the option, (ii) the number of industry respondents who did not pick the option, (iii) the number of OSS respondents who picked the option, and (iv) the number of OSS respondents who did not pick the option. Next, we apply Fisher's exact test (Fisher 1922) to the contingency matrix to compare proportions of answers of industrial vs. OSS respondents. In this way, for each question, we perform Fisher's exact tests as many as there are answer options. To control the false discovery rate incurred by multiple comparisons we adjust p-values using the Benjamini-Hochberg correction (Benjamini and Hochberg 1995).

3 Study Results

This section aims at addressing the research questions formulated in Section 2. For each RQ we first report findings of our interviews since we used them to define the survey, and then we report the results of the survey respondents distinguishing between industrial and open source participants.

3.1 Participants' Demographics

Although demographics questions were optional, all participants answered them. All *interview* participants have more than eight years of experience in software development. They mainly use Java (6) and C (5), together with some scripting languages (3). All of them use an IDE: specifically, six of them rely on *IntelliJ IDEA*, three use *Eclipse*, two use *Visual*

Studio .NET, two *PyCharm* and one *Atom*.[3] As regards the role in the company, four of them are developers, three are involved in planning, development, and testing tasks, two are researchers, and one is a project manager. One of the interviewees (I_{10}) is employed by an open source foundation, and therefore we treat this interview as representative for OSS.

Concerning the *survey* respondents working in industry (referred as Ind_n), 17 have more than 10 years of development experience, 21 between 5 and 10, while only 14 declare to have less than 5 years of development experience. Almost all of them (42) are developers: other respondents are DevOps specialists, project managers, a researcher, a software architect, a consultant, a data scientist and machine learning specialist, and a tester. The organizations they work for are software development consulting (12), data and analytics (10), information technology (10), software as a service (5), safety-critical systems (4), software for energy environments (3), and one for each of the following categories: systems programming, game development, health care or social services, government, hardware, financial and banking, and industrial automation.

Among the respondents contributing to OSS projects (referred as OSS_n), 31 have more than 10 years of development experience (63%), 11 between 5 and 10, and 7 have less than 5 years of experience. Most of the respondents own at least a bachelor degree (43): 12 have a Ph.D., 16 a Master Degree, 14 a Bachelor Degree and 1 has two Bachelor Degrees. Considering the number of open source projects to which they contributed so far, 8 declared they contributed to only one project, 15 between one and five projects, 14 between five and ten while 10 contributed to more than ten. All the respondents contributing in OSS use at least an IDE/editor: the most used ones are *IntelliJ IDEA* (21), *Visual Studio .NET* (14), *Vi* (13), *PyCharm* (11), and *Eclipse* (9). 30 respondents use two or more editors, with the largest number of IDEs reported being five.

3.2 RQ_1: To What Extent Developers Admit SATD?

We start by investigating whether developers annotate any design and implementation choice, and then we focus on the annotations highlighting the presence of TD.

3.2.1 How Frequently Do Developers Annotate Design Decisions?

Four of our interviewees report that they always annotate their code with decisions made, to improve the awareness towards the whole development team. Other four indicate that they mainly use annotations for planning purposes, e.g., I_5 uses annotations very frequently, in particular "to clarify what has to be done to the ones who will implement the code." Three of them highlight that undocumented code cannot be merged in the stable release branch due to company policies. Finally, I_8 never uses annotations since the code should be self-explanatory.

Based on the interview responses, while defining the survey we provided four alternatives: never use annotations, use annotations in less that 25% of development tasks, between 25% and 75% of development tasks, and in more than 75% of development tasks. Figure 1 suggests that OSS developers (yellow) are more likely to annotate design and implementation decisions compared to industrial practitioners (blue): only 13% of industrial participants use annotations in more than 75% of their development tasks as opposed to 37% for OSS. The difference between the two distributions is marginally significant (Wilcoxon

[3]The sum exceeds the number of interviewees as multiple IDEs and programming languages may be used.

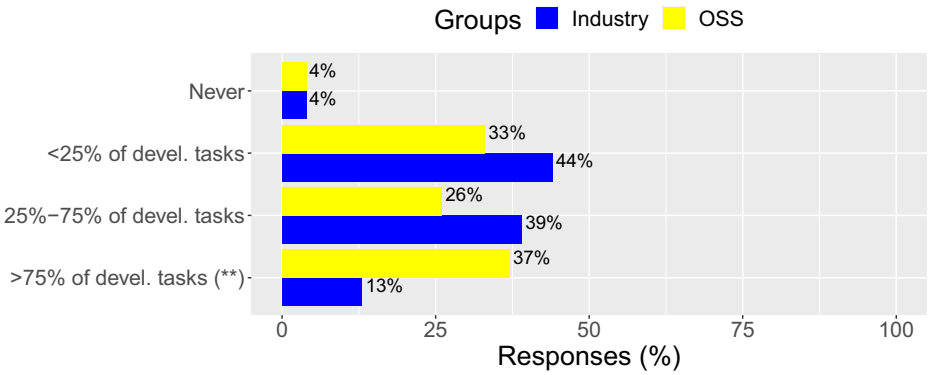

Fig. 1 Frequency of design/implementation decision annotations (Q_1)

rank-sum test p-value=0.05) with a small effect size ($d = 0.21$). The above result is slightly contradicting what we found during our semi-structured interviews from which it was quite evident that in the industry there may exist strict policies for annotating both design and implementation decisions especially for planning purposes (I_5).

3.2.2 How Frequently Do Developers Make Satd Annotations?

Most of the interviewees (7) use SATD-related annotations very often or always. However, two of them report that they try to fix the problems before considering leaving an annotation somewhere. As for the previous question, two interviewees highlight the presence of company policies prohibiting merging buggy or not fully implemented changes. For instance, I_{10} states: "We have a very strict policy that we do not add any of these comments into the source code, we try to only merge code that is fixed", or I_5 mentions "we try to only merge code that is actually fixed". About the above aspect we found contradicting results since for instance I_3 reports that "Very often it happens that I need to add annotations related to a new discovered bug", or I_7 who mentions "The frequency of FIXME is greater than the frequency for TODO. Indeed, it is more likely to find a bug in the system that you cannot fix immediately rather than pushing an empty or not completed functionality".

The survey (Fig. 2) confirms the observations coming from the previous question dealing with annotations for design and implementation decisions. 71% of the industrial respondents report that they never or rarely use this kind of annotation, while only 10% use them very frequently as opposed to 47% and 20% for OSS participants. We conjecture that in the industry it is possible that people are not very happy admitting that something is going wrong and let the other team members know about this misbehavior. As an example, one of our interviewee (I_2) states: "I am shy reporting that my code is not in the right shape, for this reason, most of the time I use documents accessible only to myself". Differences between the results are statistically significant (Wilcoxon rank-sum test p-value=0.02) with a small effect size favoring OSS respondents (Cliff's δ=0.24), i.e., OSS developers annotate delayed or intended sub-optimal activities more often than their industry peers.

Comparing Figs. 1 and 2, one might get the impression that SATD is less likely to be annotated than design decisions. More careful statistical analysis, however, reveals that while there is no statistically significant difference between the frequency of annotations for design decisions and SATD for OSS developers (p=0.11), industrial developers seem to

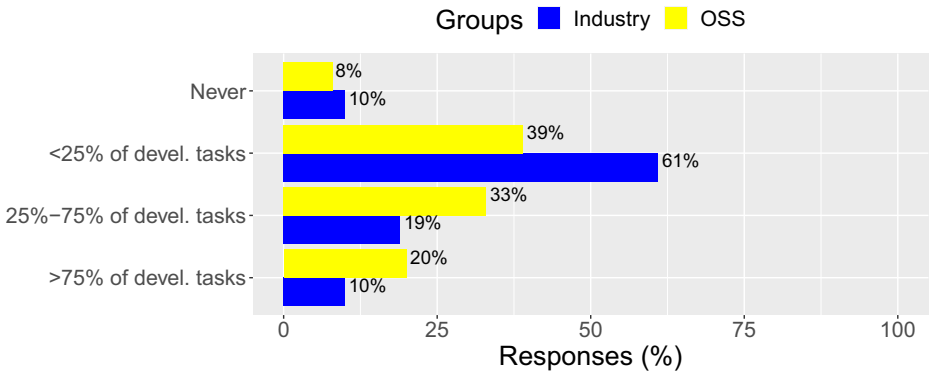

Fig. 2 Frequency of SATD annotations (Q_2)

be more reluctant to admit TD than to annotate their design decisions (p-value=0.02, Cliff's δ=0.24, small), as previously conjectured and highlighted by I_2.

3.3 RQ$_2$: What are the Reasons for (not) Admitting SATD?

Next, we investigate the reasons why developers use or not use annotations to record implementation and design choices, including technical debt. Two of the reasons have already been mentioned in Section 3.2: the presence of company-specific policies and requirement the code to be self-explanatory rendering annotations unnecessary.

3.3.1 Why Developers Use Annotations?

We perform card sorting on the interview transcripts, and three reasons emerge.

First, annotations can serve as *a reminder for the community, the development team, or the whole organization* (9 interviewees), "to make the other aware of the problems so that who wants can try to fix and address the problem once available" (I_8). This is confirmed by the survey results (Fig. 3): more than 67% of the respondents, both from the industry and from the OSS, use annotations for improving the awareness of the whole development team, project or organization.

Second, annotations can serve as *a reminder for the developer themselves* (6 interviewees). For instance, I_{10} states: "TODOs are used to remind me of necessary work before it is ready. I am usually the only one who sees them." More than 73% of the survey respondents use annotations to this end, e.g., when there is not enough time the annotations can support recollection of what has been done and what remains to do. Our distinction between annotations being used as reminders for self, team, and community concurs with the earlier observation of the annotation practices in the Eclipse community (Storey et al. 2008).

Third, annotations can be specifically intended as *a reminder for newcomers* joining the project (5 interviewees). For instance, I_6 uses annotations because "this makes it easy for the new owner of the project, for the new developers that are working on that project.", or I_3 states: "we will use annotations in order to improve the overall readability for simplifying the integration of newcomers in the team". However, more generally speaking, I_5 mentions that he uses annotations " to give the possibility to improve and understand the code later

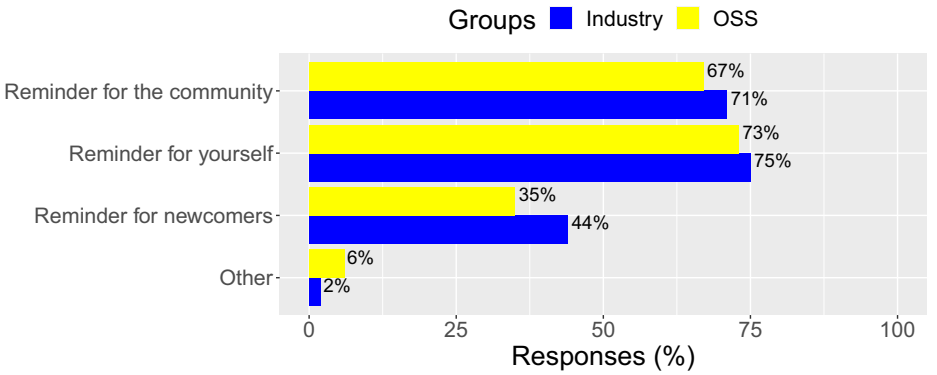

Fig. 3 Reasons behind using annotations (Q_7)

on". The on-boarding of the newcomers is mentioned by more than 35% of the survey respondents.

In addition to the three reasons that emerged from the interviews and were confirmed by the surveys, one industrial participant (Ind_6) reports that there are cases where the annotations are used to keep track of the creative development process, while one OSS developer (OSS_{23}) highlights that "code must be expressive enough to communicate its intent".

3.3.2 Why Developers Do Not Use Annotations?

Three interviewees indicate that annotations are useless when the code is "very clear, complete and works as expected" (I_1). Two themes were reported by a single interviewee: (i) reluctance to report that the code is not in the right shape (I_2), akin to the discussion of bounded transparency by Storey et al. (2008); and (ii) omission of annotations due to lack of time (I_5), likely introducing documentation issues (Aghajani et al. 2020; Aghajani et al. 2019; Arnaoudova et al. 2016). Even if developers need good documentation for maintaining and evolving a software system, previous work by Aghajani et al. (2019) found that documentation is affected by issues such as insufficient and inadequate/obsolete content. In this case, not annotating design/implementation choices and TD may generate a documentation that is not up-to-date (i.e., obsolete content) or incomplete.

Results of the survey (Fig. 4) show that lack of time is by far most common both in the industry (62%) and OSS (43%). The percentage is unsurprisingly higher for industry than for OSS, as there is likely more pressure in releasing the software. As already highlighted, it may be of interest to investigate how the not usage of TD annotations impact the overall quality of the documentation in terms of introduction of documentation smells (Aghajani et al. 2019). About 10% of the respondents state that they do not need annotations to recall where the source code needs improvement, while reluctance to admit that the code is not (yet) right has been reported by 4–6% of the respondents from OSS and industry respectively.

20 out of 52 industry participants and 19 out of 49 OSS participants provided further insights when answering this question. Three industrial and three OSS respondents are concerned by annotations becoming obsolete and confusing when the source code evolves. For instance, Ind_{32} reports: "Documentation tends to diverge from the code it refers very quickly. If you change the code then you should change the documentation accordingly and

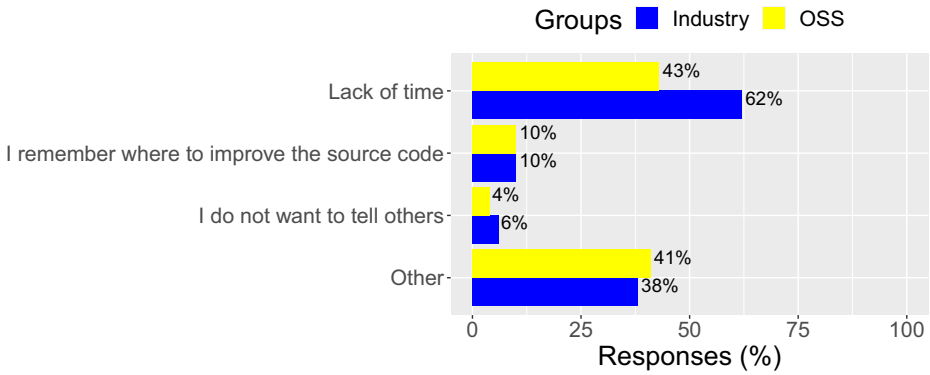

Fig. 4 Reasons behind not using annotations (Q_8)

it rarely happens, meaning that your documentation easily becomes misleading". Moreover, according to OSS_{23}: "<i>n the very best case annotations are only redundant information. In the worst case they are not in sync and don't communicate the true story. Truth can only be found in code." The latter confirms what already reported by previous literature, indeed, Wen et al. (2019) in their taxonomy of code-comment inconsistencies identified three different type of inconsistencies dealing with TD (e.g., SATD comments not deleted while addressing them, or TD introduced without being documented).

Also in this case differences between the responses of the industry and OSS participants are not statistically significant.

3.3.3 Influence of Organizational Policies

As mentioned above, several interviews suggested that organizations might have policies governing when annotations should, can or should not be used. Hence, we included in the surveys two open-ended questions about policies on (not) annotating (i) design and implementation decisions, and (ii) SATD. Such policies are summarized in Fig. 5 (blue and yellow boxes indicate the number of responses for each policy provided by industry and OSS respondents respectively).

When asked about the policies for annotating development decisions, 32 respondents from the industry and 29 respondents from OSS reported that their organizations or projects do not have such policies. Applying card sorting to the remaining answers we identify seven themes:

- *Never annotate*. Six industry and three OSS respondents indicate that the source code must be self-explanatory rendering annotations unnecessary: "following clean code principles and investing a lot of time into good naming of variables and functions is key" (OSS_{23}). The latter is confirmed by an industrial developer (Ind_{32}) who points out that: "if the design needs explanation then it is probably wrong and needs to go back to the drawing board.".
- *Always annotate*. On the other end, we find five industry and two OSS respondents reporting that annotations are mandatory, e.g., "A Public procedure must be tested and documented." (Ind_6) or "10% of each day is used to document the days' work." (OSS_{44}).

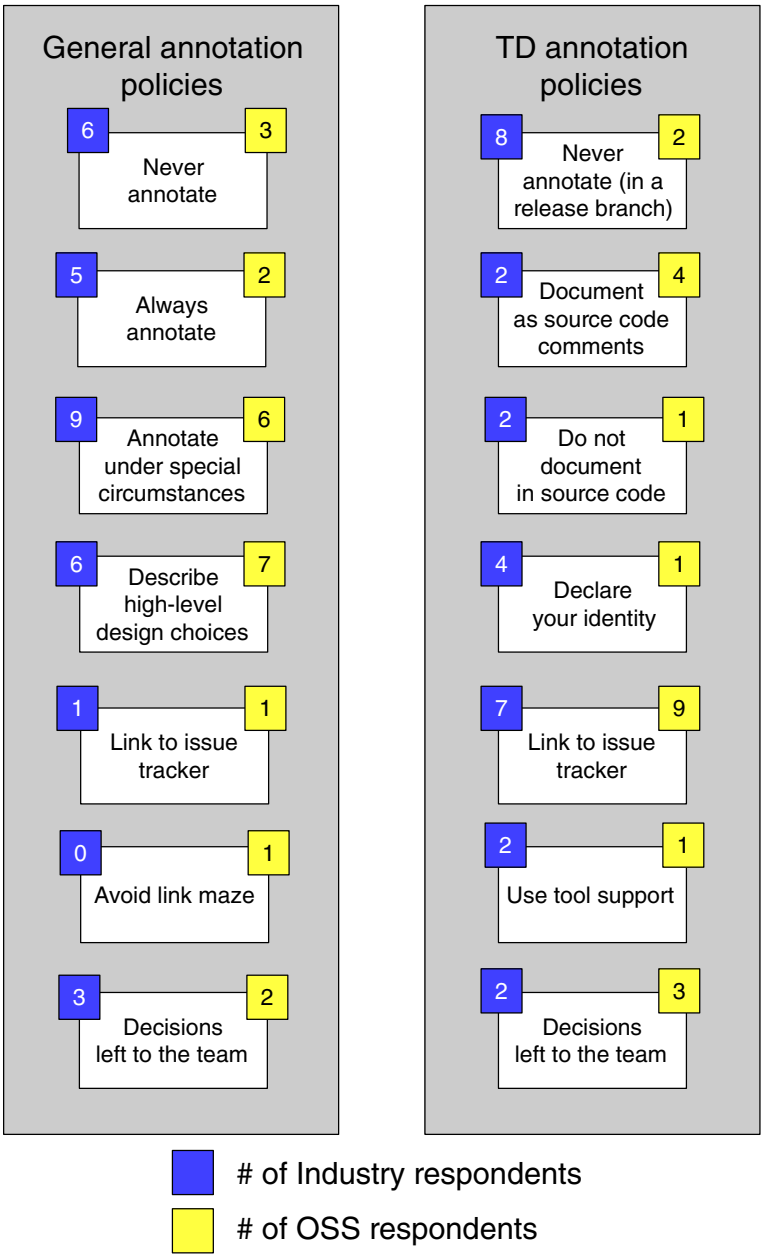

Fig. 5 Summary of general and TD annotation policies (Q_3 and Q_4)

– *Annotate under special circumstances*. According to nine industry and six OSS respondents, annotations are only mandatory when specific conditions are met: (i) when the code is not self-explanatory, e.g., "for some very dark and obscure snippets of code that

cannot be refactored for better clarity" (Ind_{32}), (ii) only in the early stage of development when presumably the developers' ideas are not yet fledged fully, and "these are gradually fixed in later commits" (Ind_9), and (iii) when the code has to pass the static analysis checks, e.g., OSS_{25} reports :"Eslint restrictions, not using case insensitive names for components etc.".

- *Annotate to describe high-level design choices* (six industry and seven OSS respondents). As an example, Ind_{45} states: "Design choices have to be documented in the model of the SW architecture or detailed design." One way to support developers in this task is to have tools automatically generating such descriptions.
- *Link to issue tracker*. Respondents Ind_{19} and OSS_{41} indicate that the commit messages should record decisions and also link to the issue tracker.
- *Avoid a link maze*. In contradiction to the previous case, OSS_5 mentions that the policies "result in a maze of links which discourages reading them."
- *Decisions left to the team*. Three industry and two OSS respondents indicate that the decision whether to use annotations is left to the development team. For instance, Ind_{19} mentions: "No policy within the organisation, but sometimes a policy within the team", however the respondents does not provide any insights indicating when and why this happens.

Moving the attention to the policies for annotating TD, 32 industrial and 30 OSS respondents state that they do not have such policies. As above, no statistically significant difference between the industrial and OSS respondents could be observed (Fisher's exact test p-value $\simeq 0.69$). We derive seven themes related to TD annotations:

- *Never annotate (in a stable release branch)*. While, for design decision, interview respondents pointed out a *Do not annotate* policy, for SATD interview participants told us this is mainly for the stable release branch (while such annotations are still possible in development branches). In some companies, this policy is automatically enforced: "Our CI/CD workflow prevents TODO, FIXME comments from proceeding past the DEV branch" (Ind_{13}). This theme confirms what we found in our interviews, indeed I_5, while answering whether or not they use TD annotations states: "We have very strict policy that we do not add any of these comments into the source code, we try to only merge code that is actually fixed."
- *Always document TD as source code comments* (two industry and four OSS respondents). This policy is the opposite of the previous two. Indeed, in this case, it is possible to push a change containing a workaround or delayed activities highlighted with a specific source code comment. For instance, Ind_3 states: "They should always be documented", while an OSS developers (OSS_{40}) mentions: "ad-hoc todo code comments".
- *Document TD in shared documents and not in the source code* (two industry and one OSS respondent): "Hacks and workarounds are discouraged but, when absolutely needed, they are usually explained not in code but in documents that are in our knowledge base" (Ind_{32}).
- *Link to issue tracker* (twelve industry and nine OSS respondents). As for design annotation, interview participants pointed out how to use comments vs. issues, and how to link them. This policy separates recording of the details of TD (in an issue tracker) and indicating presence of TD in the source code (by including the link to the issue tracker): "ToDos are fine, but generally, future work shouldn't be documented in code instead of in tickets" (OSS_7). In a previous study with Eclipse developers (Storey et al. 2008),

44% of the respondents indicated that they include bug ID when using TODOs. While this can be further investigated and quantified by mining and analyzing SATD comments in source code, a preliminary exploration of the SATD comments in the dataset by da S Maldonado and Shihab (2015) reveals that in OSS developers do not seem to follow this practice.

– *Use of tool support* (two industry and one OSS respondents). This policy is induced by static analysis tools detecting TD, and requiring developers to either resolve TD or at least justify its presence before committing the change. For example, one of the survey participants (Ind_{21}) highlights: "The code is scanned by Sonar and warns you that you're committing a todo. You have to justify the existence of the TODO's to code reviewers.".
– *Declare your identity when introducing TD* (four industry and one OSS respondent). Some policies require developers to take personal responsibility by disclosing their identity in the SATD comment: "no TODO without names or initials of the engineer" (Ind_{21}). In the aforementioned study of Eclipse (Storey et al. 2008), 51% of the respondents reported including their name or initials in TODO annotations. Also in this case, a preliminary exploration of the SATD comments in the dataset by da S Maldonado and Shihab (2015) reveals few cases where developers disclose their identity while introducing an annotated TD.
– *Decisions left to the team* (two industry and four OSS respondents). Similarly to design decision annotations, TD annotations can be defined by individual teams.

3.4 RQ$_3$: What are the Channels and Tool Support used to Admit TD?

Section 3.3.2 suggested that respondents tend to use multiple channels to record SATD: in addition to the source code comments, interviewees and survey respondents have mentioned issue trackers and private documents. Moreover, a recent study by Xavier et al. (2020) conjectures that developers can admit TD by not only using source code comments, e.g., by creating issues in tracking systems and labeling them as referring to TD. Their results highlight that only 29% of the TD admitted in issue trackers can be tracked to source code comments pointing out the lack of overlapping among what documented in source code comments and what highlighted in issues.

Zooming in on this aspect during the interviews, we found that the use of source code comments is widespread (7 out of 10), but that developers also use issue trackers (4), internal mailing list (3), task management tools, e.g., Trello (3), private documents (1) or commit messages (1). Hence, we used the following as answer options in the surveys: source code comments, issue trackers, internal documents, task management tools, mailing lists and commit messages. We have also included code reviews or pull requests and messaging apps, e.g., Slack.

Figure 6 shows that industrial and OSS respondents use different tracing mechanisms in very similar ways (differences are not statistically significant). Similarly to the interviews, source code comments are the primary channel for communicating design decisions and TD, being used by more than 81% of respondents. Commit messages, code reviews or pull requests, and issue trackers are still very common in both industry and OSS (44% of respondents). Finally, 44% of industrial respondents and 35% of OSS developers use internal and private documents to keep track of design decisions and TD. This may or may not provide awareness to everybody in the development team, and surely not to external contributors/newcomers of OSS.

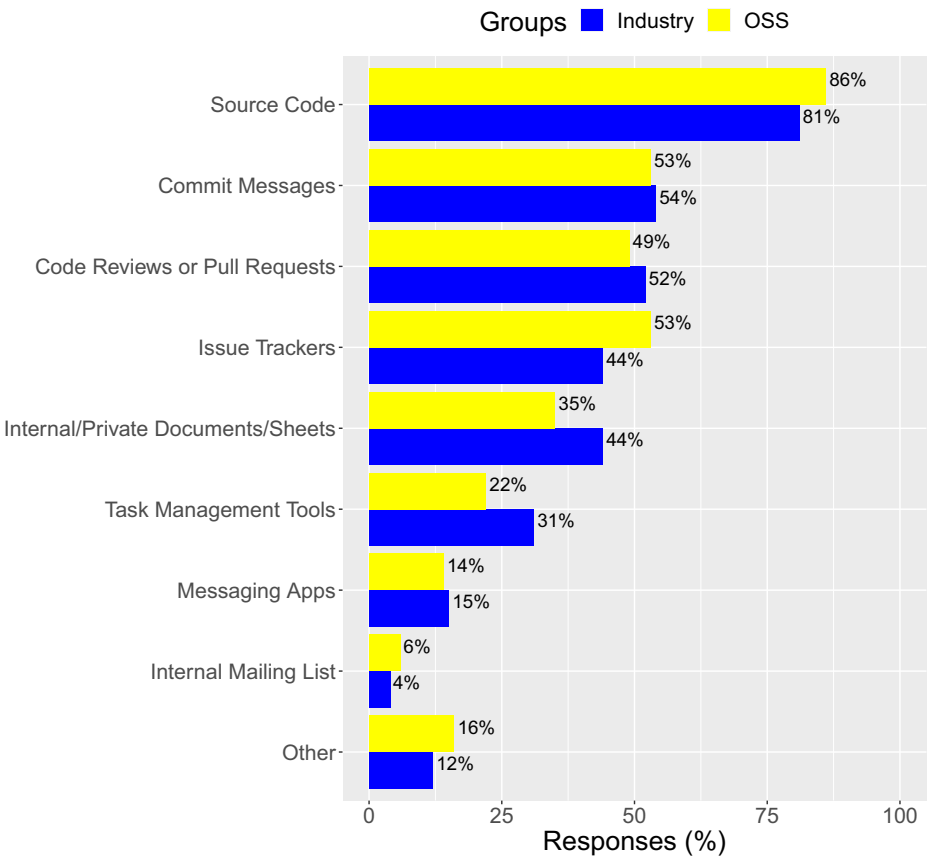

Fig. 6 Tracing mechanism (Q_5)

3.5 RQ_4: What is the Content of SATD Annotations?

By applying card sorting to the interview transcripts we identify seven topics. The topics were also used as answer options in the survey (cf. Fig. 7).

- *Explain the need of improving maintainability or performance* (six interviewees), e.g., I_3 explicitly refers to the need to improve "readability of the code under development". The survey shows that mostly maintainability (63%) and performance (35%) need to be improved.
- *Explain the presence of a bug* (six interviewees): adding information related to the bug such as the context in which it occurs, the steps to reproduce it, or the observed behavior opposed to the expected one. In this context, I_6 highlights: "What are the next steps and what was the source of problem/delay (only technical problems are included)". Survey respondents mentioned this reason in a minority of cases (27-29%), likely because it may be debatable whether or not bugs should be considered TD (Li et al. 2015) or whether they should be handled in the usual way (i.e., through issue tracking/triage).

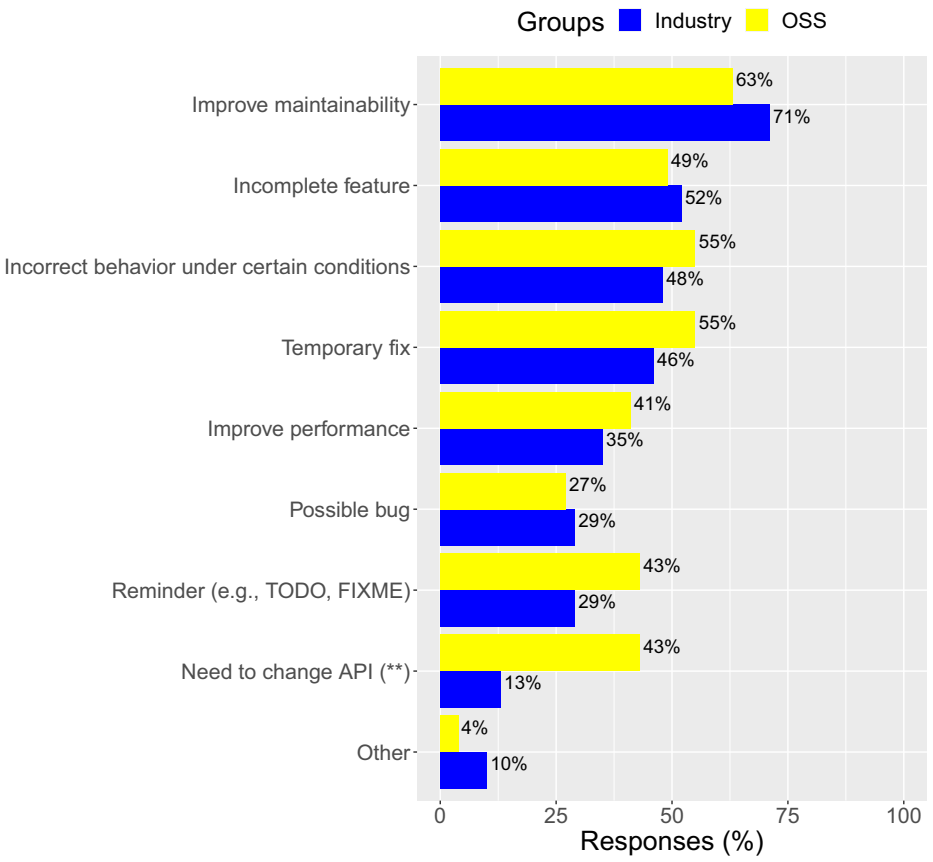

Fig. 7 Type of content added while annotating technical debt (Q_9)

- *Report that a feature is not ready yet* (five interviewees). I_2 also stresses the importance of "explaining what remains to do". This is also considered important by most of the survey respondents (49-52%).
- *Report the need for introducing a temporary fix* (three interviewees). For example, I_1 states: "... in presence of a bug into a different feature I am using, I can complete my implementation but considering that this is only a temporary patch that needs to be modified once addressed the inherited bug". This observation has been confirmed by more than 46% of the survey respondents.
- *Need for using a better API/upgrade API* (two interviewees), e.g., when developers "have no time to find a better API" (I_1). This is the only case for which OSS survey respondents considered the option in significantly more cases (43%) than industry (13%). We conjecture that OSS projects are more likely to use a variety of third-party APIs, whereas industrial projects might be subject to organizational policies or technological constraints.
- *The code works only under specific conditions* (two interviewees). This observation has been confirmed by more than 48% of the survey respondents.
- *Use a simple tag as reminder* (two interviewees). I_8 states: "... when I am sure that I will touch the same code in the next day, I only use a tag because I will remember

also without adding more information". The survey indicates that a limited number of developers use tags as reminders, especially in industry (29% vs. 43% in OSS). Once again, as mentioned above, the limited usage of such tags may depend on the reluctance industrial developers have to expose the limitations of their source code. This use of SATD annotations is similar to short-term task annotations discussed by Storey et al. (2008).

Among the six survey respondents that used the "Other" option, Ind_{34} indicated that SATD may refer to (partially) stubbed functionality: "Listing missing functionality in stubs. For example when implementing a bigger feature and you create multiple stubs first to have all the required functions." This is a specific case of a feature not being ready yet.

3.6 RQ_5: How do Developers React when they Encounter SATD Comments?

The goal of our last question is to investigate what developers usually do while encountering a previously admitted technical debt while coding. Most of the interviewees (7) reported that, when they encounter a SATD annotation, they try to address it before completing their tasks, even if the annotation was left by somebody else. Three interviewees do not address the TD, but take it into account while performing the changes. Quite surprisingly, two interviewees report that they usually comment out the annotations and complete their tasks without accounting for it. More specifically, I_1 states: "There are also cases where I comment out the code affected by the annotations and I will try to complete my task ignoring it". Finally, four interviewees ask the project manager or the author of the admission before starting to work on the affected piece of code. As an example, I_2 states: "I try to contact the original author in order to have more insights and only when the original author cannot address the content of the annotation, I will try to address it by myself".

Our survey results (Fig. 8) are in-line with the interviews, and there is no statistically significant difference between industry and OSS. Many respondents try to find a solution either when the SATD comment was added by them (54% of the industrial respondents, and 65% of the OSS respondents), and when they are not (40% of the industrial respondents, and 39% of the OSS respondents). Furthermore, 38% of the industrial and 43% of the OSS respondents report to not address the technical debt but to consider it while completing their

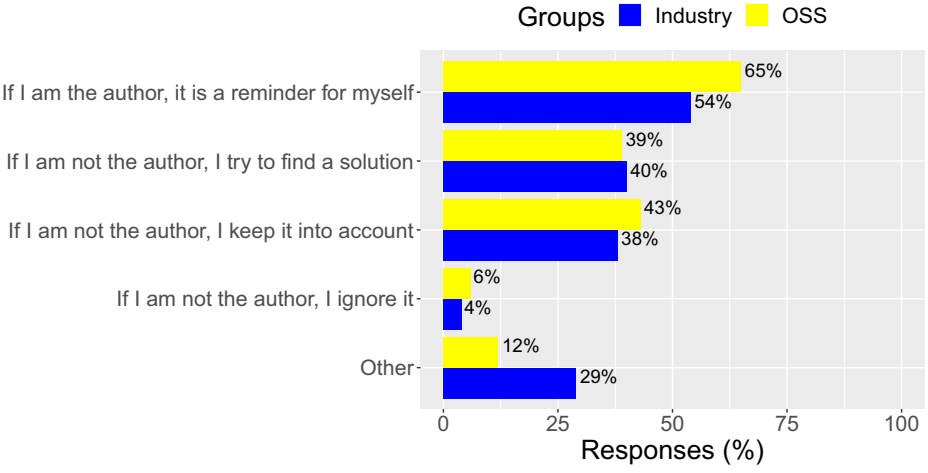

Fig. 8 How developers deal with SATD annotations (Q_{10})

tasks. Only a small minority (4% industrial and 6% OSS) completely ignores the admissions. The "Other" options include further reasons influencing the decision to address TD. Ind_9 mentions priorities: "Depends on the priority of the task. Delivering results is more important than over-engineering something that works assuming there are no major flaws with the code." Ind_{22} comments about code ownership/familiarity: "If it is in parts that I touch a lot, I would be more inclined to try to find a solution. If not, mostly I would ignore the comment".

4 Implications

In the following, we summarize the main lessons learned from this study, which can result in implications for practitioners and, in some cases, for researchers. In the following, after discussing results, we state some possible implications that emerge from such results. Then, we outline possible research directions that would help to enact the implications.

Industrial and OSS Developers Annotate Design and Implementation Decisions with a Similar Frequency, OSS Developers Admit Satd More The results of RQ_1 suggest that industrial and OSS developers annotate design and implementation decisions with a similar frequency. At the same time, industrial developers seem to be more reluctant to admit SATD: survey respondents attribute not admitting TD primarily to lack of time (RQ_2). As explained in Section 3.3.2, this is a primary reason for not admitting TD for both industry and OSS, although with a greater percentage in industry. On the one hand, the reason may be found in the higher pressure industrial developers have, as they need to prioritize software releases, e.g., by adding requested features or fixing critical bugs. On the other hand, this might or might not be the primary reason for lack of admission, because respondents might be more comfortable to indicate lack of time than "not being able to do their job right". As reported in Section 3.2.2, one interviewee (I_2) stressed this aspect clearly reporting the shyness as a main factor for not annotating TD in the source code while using private documents that cannot be accessible to other team members. Moreover, some companies have specific rules for which "not ready yet" code should not be merged into a master/stable branch or should not be annotated in the source code but elsewhere.

Implication 1a: To reduce the barriers to manual annotation of TD, tools should be able to recognize situations where TD should be admitted.

Previous research has shown how scenarios where design TD should be admitted could be identified through machine learning models that leverage source code metrics, static analysis warnings, or code smells (cf. TeDIOUS (Zampetti et al. 2017)). Future research should better explore these scenarios, by recommending admittance of other types of TD, as well as helping developers in understanding why the code is likely to contain TD which should be, therefore, admitted.

Implication 1b: Companies and OSS projects should find a balance between the desire to ensure quality and promoting an "open" culture in which the awareness of temporary solutions, incomplete, and in general not ready components is encouraged and not considered as a bad practice.

The latter is mainly a cultural and organizational problem rather than a technological one. Promoting a knowledge sharing culture also for negative aspects of the source code is something to encourage. Project managers should avoid scenarios where developers may fear of bad consequences of admitting their code limitations.

SATD is Mainly Communicated Through Source Code, While in a Few Cases Internal Documents are Used While source code remains the premier channel to communicate SATD, some developers (both from industry and OSS) mention commit messages, code reviews or pull requests, and issue trackers as well as internal documents. The latter should be avoided, as it reduces the problem awareness and knowledge sharing, especially in open source projects where external developers may want to contribute. Based on our results, the use of other, internal documents to convey SATD is more prevalent in industry than in OSS. Instead, the use of source code comments represents a very transparent channel of communication, with a sufficiently clear tracing of the problem description onto the affected source code elements. At the same time, source code comments are highly unstructured and usually do not provide a notification mechanism (Guzzi 2012; Storey et al. 2006).

Implication 2a: Researchers should provide mechanisms to help developers write standardized, easy to understand SATD comments, as well as enacting a notification mechanism similar to code review tools or issue trackers.

For example, it could be possible to use tools similar to the TODO bot (Etco 2017) that creates GitHub issues based on TODO annotations in the source code. Such tools could be combined with SATD detection tools (da S Maldonado and Shihab 2015; da S Maldonado et al. 2017b; Huang et al. 2018; Ren et al. 2019), in order to create issues also for SATD not as easily recognizable as TODOs.

Implication 2b: Researchers should expand studies of SATD beyond the source code.

In this context, Xavier et al. (2020) have studies how SATD is admitted in issue trackers. However, based on RQ_3 results, TD is admitted on a variety of channels, which should also be considered to gain a global picture of a project's (explicit) TD.

Developers Believe to Remember What Needs to be Improved. As a Consequence, the Decisions' Rationale is not Tracked, and this Could Create a Truck Factor Annotations are used for both industry and OSS developers as a reminder for themselves and others, less so for newcomers that can be expected to benefit most from recorded design decisions or TD indications. Moreover, developers still believe that, if they remember what to do, TD annotations are redundant or confusing. Reliance on one's memory can create multiple problems including lack of traceability and documentation of decisions taken (Aranda and Venolia 2009; Falessi et al. 2013; Alexeeva et al. 2016). Not only this makes the life hard for project newcomers, but, in extreme cases, it can provoke truck factor events (Avelino et al. 2016; Torchiano et al. 2011), making the source code hard to be maintained or project becoming abandoned (Avelino et al. 2019). Missing rationale is also frequently reported as confusing during code review (Ebert et al. 2019).

Implication 3a: Even if a developer is aware of what needs to be improved, consistently adding SATD and documenting decision rationale helps to promote awareness in the project, in turn, supporting newcomers and reducing the likelihood of project abandonment.

For this reason, having the complete picture of a project's SATD should contribute to the set of (documented) design and development decisions that one may want to look when understanding a project, for example in the case of newcomers starting to contribute in project's activities.

Implication 3b: Tools should support co-evolution of annotations with the source code as well as automatic documentation generation and possibly automatic machine translation from source code to comments.

Previous work has pointed out how sometimes source code comments may be misaligned with source code (cf. Aghajani et al. 2020; Wen et al. 2019) and such inconsistencies could be highlighted to developers (Tan et al. 2012). In the context of SATD management, SATD comments could be automatically updated (or even removed) when the source code is changed. Also, similarly to approaches for commit (Jiang et al. 2017) or release note generation (Moreno et al. 2017), SATD comments could also be automatically generated.

SATD Annotations may be Beyond Maintainability and Temporary Fixes Most of the SATD annotations contain details about maintainability concerns and temporary fixes (cf. Fig. 7). Sometimes the annotations refer to the functional correctness; whether bugs should be considered as TD is controversial: e.g., Bellomo et al. (2016) exclude bugs from TD, while Li et al. (2015) consider bugs as a special case of TD. In any case developers should be encouraged to report the problem as a bug in the issue tracker and properly handle it. Still, there may be corner cases where the undesired behavior is considered acceptable, e.g., it only happens in rare cases and the harm caused is limited.

Another specific case of annotation, especially mentioned by OSS respondents, was about the need for API replacement. While we do not have any evidence of why API changes were mentioned more by OSS developers than by industrial developers, one possible interpretation is that the former have more degrees of freedom in integrating third-party, non-certified components in their software.

Implication 4a: SATD may be related to functional behavior, and not only to maintainability problems.

Certainly, when we mention functional behavior, this is not about blocking bugs but rather, from what we have observed, anomalous behavior in certain (sometimes rare) circumstances. On the one hand, the latter enforces the need (as pointed out by Implication 2a) to manage SATD with issue tracker. On the other hand, testing and analysis activities that identify (unlikely) scenarios where the program may not work as expected could also automatically generate SATD comments.

Implication 4b: Recommenders supporting developers in the choice of APIs could help solving SATD related to API upgrade.

To help resolving this kind of TD, recommenders, and specifically those recommending APIs based on their non-functional properties (e.g., ease of use, performance, security) (Lin et al. 2019; Uddin and Khomh 2017) could help to address this kind of SATD. While,

as explained in Section 3.5, this kind of TD is more prevalent in OSS than in industry, recommenders similar to those mentioned above could be adopted in both contexts indistinctly.

In Most Cases, SATD is Taken into Account or Even Addressed, Although Sometimes Developers "hide" it Our results indicate that more than 40% of developers generally take SATD into account, or try to address it, either if it is coming from themselves (i.e., they have introduced the SATD annotation when writing source code) or by others. This being said, many developers merely take SATD into account without trying to address it: this is concurrent with the previous observation of da S Maldonado et al. (2017a) that while all respondents of their survey at least sometimes encounter SATD, half of them rarely address it. In a few cases, developers just decide to ignore SATD: this concurs with an earlier observation of Zampetti et al. (2018) that 2–17% of SATD instance removals do not involve modification of the method source code. While in some cases this may be a legitimate behavior (i.e., the problem does not manifest anymore, or the concern no longer applies because of changes elsewhere), there are also cases where this should be considered as a "smelly" behavior.

Implication 5: Hiding SATD without addressing it should be discouraged.

To this aim, researchers could develop tools to recognize such behavior similarly to what, for example, has been done for Continuous Integration pipelines, to discourage "hiding" failing test cases or static analysis checks (cf. CI-ODOR Vassallo et al. 2019). Also, it would be desirable to develop recommender systems that prioritize actions to be undertaken in the presence of SATD.

5 Threats To Validity

Threats to *construct validity* concern the relationship between theory and observation. One threat can be due to the lack of a clear separation between industrial and OSS developers, i.e., some industrial developers may also participate in OSS during their spare time or their company contributes to OSS. However, we know that the industrial developers coming from our personal contacts answered the survey (or the interview questions) based on the work carried out in their industrial projects.

When sending the questionnaire to OSS developers, by using the mailing lists of the projects, we asked the respondents to answer with respect to that particular OSS project. That being said, it is entirely possible that developers working on both industrial and open-source projects may have a slightly different perspective in admitting TD than developers working only on OSS or only on industrial projects.

Another threat to construct validity can be due to the interpretation the study respondents have given to the notion of TD and SATD. As for the interviews, we clarified with the participants the intended notions referring to the concepts of previous studies (Bavota and Russo 2016; Potdar and Shihab 2014). As for the questionnaires, we accompanied it with an introductory letter (available in the replication package (Zampetti et al. 2021)), and made sure to have self-explanatory questions. That being said, we cannot exclude possible cases of misinterpretation. In particular, since our questionnaire contains specific questions about TODO- and FIXME-related comments, some respondents may have assumed that any

comment with such an annotation is TD-related. However, based on Cunningham's definition (Cunningham 1992), this may or may not be the case.

Finally, a further threat can be the use of multiple-choice questions for the survey, which was chosen to ease data collection and minimize the respondents' burden. To mitigate this threat, we allowed respondents to use the "Other" option and provide a written answer when the options did not fit.

Threats to *internal validity* concern factors that are internal to our study that could have influenced the results. One threat could be the low representativeness of the respondents with respect to our initial target. As for the industrial developers, relying on personal contacts limited this threat. As for the OSS developers, to at least achieve diversity in terms of programming languages, we tried to balance the number of respondents among the different languages. Moreover, by contacting the mailing lists of 104 OSS, and by involving developers of small, midsize and large companies, we tried to achieve diversity in terms of organizational culture. Of course, it may be desirable to diversify other dimensions as well.

It could be possible that the experience of the participants may affect our results. To mitigate this threat, we applied the same statistical procedures applied to compare industry and OSS developers to check whether there are statistically significant differences between developers with less than 5 years of experience and those having more than 5 years of experience. The results did not indicate, in any case, any statistically significant difference (p-value always > 0.05).

Similarly, it could be possible that OSS answers depend on the programming language of the targeted project. We have tested the effect of the language, for Q_1 (Fig. 1) and Q_2 (Fig. 2) using Kruskal-Wallis test (Kruskal and Wallis 1952), and for the other questions using a proportion test (Newcombe 1998), adjusting p-values for multiple comparisons using the Benjamini-Hochberg correction (Benjamini and Hochberg 1995). Results, likely because of the relatively low number of responses for each language, are never statistically significant (p-values> 0.05), except for Q_7 (Fig. 3), where we found that, for Python, nobody mentioned the use of annotation as a support for the community.

Threats to *conclusion validity* are mainly related to the use of analysis techniques to support our findings. As explained in Section 2, we used suitable statistical procedures and effect size measures to answer our research questions and above all to compare responses coming from the industry and OSS. Moreover, subjectiveness might have affected the coding of the interview transcripts and open-ended questions. Using multiple annotators and multiple rounds of coding mitigates this threat.

Threats to *external validity* concern the generalization of our findings. Clearly, the results may not generalize beyond the companies and the OSS to which the survey participants belong. We mitigated this threat having a set of industrial developers working with different languages, as well as receiving answers from mailing lists of OSS written in different languages.

Last, but not least, it is important to remark that our study concerns SATD in source code. This may or may not cover a broader definition of TD. However, TD management has been largely investigated in other studies, as discussed in Section 6.1.

6 Related Work

This section reports the literature related to (i) studies about TD, (ii) "self-admitted" TD and (iii) source code annotations.

6.1 Studies on Technical Debt

Previous studies discussed the term "technical debt" (Brown et al. 2010; Kruchten et al. 2013; Seaman and Guo 2011) underlining that TD is mainly used for communication between developers and managers for development issues. Alves et al. (2014), defined an ontology for TD, identifying 13 types of TD, related to architecture, build, code, defect, design, documentation, infrastructure, people, process, requirement, service, test automation, and test. With respect to their ontology, our study mainly focuses on source code-related TD and aims at investigating the reasons for admitting it (or not).

Zazworka et al. (2011) studied the impact of design TD on the quality of a software product highlighting the need for identifying and managing them closely in the development process to reduce their negative impact on software quality.

Lim et al. (2012) conducted interviews with 35 practitioners to understand their perception of TD. Among other insights that emerged from the interviews, there was the need for properly communicating about TD, both towards the management (to be recognized about the effort in dealing with it), as well as the customers. Surprisingly, while there could be benefits from developers' perspective to document TD, in our study we also found that both developers are afraid of admitting TD (as they fear they are under-performing), and organization guidelines discourage SATD in the main project branch. We believe that different beliefs can also be due to cultural reasons.

Ernst et al. (2015) conducted a study with professionals, combining surveys and semi-structured interviews, shedding the light on (i) the common understanding of TD, (ii) to what extent TD relates to architectural choices, and (iii) tools being used to manage TD. We believe our study is complementary to the work of Ernst et al., as we study how developers document TD. At the same time, our study relates to many of their results. In particular, 79% of Ernst et al. respondents indicated that the lack of TD awareness is a problem. Moreover, concerning tools, while we share with their results the use of issue trackers as one way to manage TD, our study reveals other kinds of communication tools being used to document SATD.

Besker et al. (2018) conducted an exploratory study aimed at understanding how software startups reason about TD. Specifically, they interviewed 16 software practitioners belonging to seven startup companies to determine what are the organizational factors that could influence the accumulation of TD, as well as, the challenges and benefits of TD for software startups. Their results present six organizational factors that lead to the accumulation of TD like developers' experience, uncertainty, and autonomy as regards TD-related decisions. Among their findings, we highlighted a quite contradictory result since they found how it is important to remove TD while joining newcomers in the development team to avoid those new developers may model their code off existing TD or duplicate poorly written code. Looking at our interviews and survey responses, we found that the admission of TD is mainly aimed at giving awareness to newcomers joining the project.

From a different perspective, de Almeida et al. (2018) conducted a multiple-case study with two big software development companies to investigate whether accounting for business objectives can improve the decision making for prioritizing TD removal, focusing more on business urgency and criticality. Their findings show how the business perspective and business processes affect TD removal prioritization. Based on these results, in a follow-up study, de Almeida et al. (2019) presented a framework for TD prioritization ("Tracy") by using a business-driven approach built on top of the business process.

6.2 Self-Admitted Technical Debt

Potdar and Shihab (2014) observed that developers tend to "self-admit" technical debt (SATD) using comments highlighting the existence of somewhat temporary. Moreover, they identified 62 patterns that indicate SATD and emphasized that the presence of SATD is not uncommon in software projects. da S Maldonado and Shihab (2015), instead, used source code comments in order to determine different types of technical debt showing that the most common type of SATD is design debt. Bavota and Russo (2016) performed a qualitative analysis of SATD and created a taxonomy featuring 6 higher-level TD categories specialized in 11 sub-categories. Similarly to our discussion of the study of Alves et al. (2014), rather than providing yet another categorization of SATD, we study the reasons developers have to admit (or not) TD and the extent to which they do that. Fucci et al. (2021) conducted a study in which SATD comments have been classified based on their content (instead of life-cycle dimensions, as previous work did). Also, they analyzed the polarity of SATD comments belonging to different categories. Finally, they studied the presence of external references in these commits. They found how on-hold and functional-related SATD are those with the most negative polarity, and that only a minority of comments leverage external references.

Different approaches have been proposed to detect SATD, including a keyword-based (da S Maldonado and Shihab 2015), text mining (Huang et al. 2018), Natural Language Processing (da S Maldonado et al. 2017b), and convolutional neural networks (Ren et al. 2019).

Rantala et al. (2020) went deeper on SATD comments containing keywords such as TODO and FIXME (i.e., KL-SATD) and compared them to the rest of source code comments. While the median percentage of KL-SATD is very low ($\simeq$ 2%), their contents is very different from other comments. They also developed a machine learning classifier for identifying KL-SATD showing good performance, i.e., AUC equals to 0.88.

Maipradit et al. (2020), instead, after a qualitative study of 333 SATD comments identified a specific category of SATD, namely "on-hold" SATD, i.e., debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere, designed an automatic approach for identifying on-hold instances having an AUC of 0.98.

Wehaibi et al. (2016) measured the impact of SATD on software development practices finding that the presence of self-admitted technical debt leads to a complex change in the future. From a different perspective, Zampetti et al. (2017) developed a machine learning approach that, by leveraging structural information (metrics or warnings raised by static analysis tools) is able to recommend developers design TD to be admitted.

da S Maldonado et al. (2017b) focused on SATD removal analyzing the change history of five Java open source projects. They found that (i) the majority of SATD is removed, (ii) SATD comments are mainly self-removed, and (iii) the survival time varies from one project to another. Moreover, da S Maldonado et al. (2017a) investigated what are the activities/reasons that lead to the removal of SATD conducting a survey with 14 developers. Their results highlighted that SATD is usually removed as part of bug fixing activities (9 out of 14 respondents indicated that) and the addition of new features (5 respondents). In a follow-up work, Zampetti et al. (2018) performed a fine-grained analysis of SATD removal, finding that a large percentage of SATD is removed accidentally while removing the whole classes and/or methods affected containing the SATD comment, and identifying six specific removal patterns. Mensah et al. (2018) clustered SATD comments for the purpose of classifying them into complex (buggy-prone) and trivial tasks, and also to estimate the amount

of change required to address the SATD, which is between 10 and 25 commented LOC for complex tasks.

Related to the work of da S Maldonado et al. (2017a), Zampetti et al. (2018), and Mensah et al. (2018), we surveyed developers investigating the extent to which they address SATD admitted by others, or keep it into account.

Xavier et al. (2020) studied the admission and removal of technical debt in GitHub issue reports. The authors have also checked the extent of the overlap between TD admitted in issue reports and source code comments being deleted while closing and fixing the above issues. Our results confirm their findings, indeed they highlighted that 71% only rely on issues to track the presence of TD in the code, while in 29% of the cases the admission was also in the source code as a comment.

From a different perspective, Fucci et al. (2020) using a curated SATD dataset, analyzed the extent to which SATD comments are introduced by authors different from those who have done last changes to the related source code and have measured the ownership of developers admitting TD on the related source code. Their results highlight that the percentage of TD admissions by developers who have not authored the last change to the related source code varies in the range [0-16], representing a non-negligible phenomenon. Moreover, by looking at the level of ownership they found how this level is not low and is dependent from the project.

6.3 Source Code Annotations

Eclipse, IntelliJ IDEA, Visual Studio and other modern IDEs support such task annotations as TODO, FIXME, and XXX (Storey et al. 2008; Storey et al. 2009; Chen et al. 2012; Padioleau et al. 2009). While SATD can be expressed using predefined tas annotations, it is often not the case: e.g., among 62276 SATD annotations collected by da S Maldonado et al. (2017b) only 3590 (5.76%) use TODO, FIXME or XXX annotations. Consistently with the previous studies (Storey et al. 2008; Chen et al. 2012) TODO is the most commonly used annotation in the dataset of da S Maldonado et al. (2017b).

Similarly to the previous studies we observed that annotations are used to create awareness (Chen et al. 2012) targeting self, team, and community as a whole (Storey et al. 2008). With respect to the contents of the annotations we take complementary perspectives: while (Chen et al. 2012) focus on the general purpose of the annotations, e.g., denoting future actions, explaining the design and providing solutions, and (Storey et al. 2008) investigate articulation work, i.e., activities related coordination and management, we take a more traditional software engineering perspective and distinguish, e.g., maintainability, incomplete implementation and temporary fixes (cf. Fig. 7).

7 Conclusion

While previous research has largely investigated the presence of self-admitted technical Debt (SATD) in software projects, little has been done to understand developers' rationale behind such annotations and reasons for admitting or not SATD. That is different developers could admit SATD more or less regularly, providing different content, and handling it differently. Moreover, external factors such as organizational or cultural ones could influence developers' SATD admittance.

To the best of our knowledge, this paper presents the first empirical study aimed at investigating SATD from developers' perspective and has been conducted through semi-structured interviews involving 10 developers (one of which from OSS, the rest from industry), and a survey with 101 participants, 52 from the industry and 49 from OSS.

Results clearly indicate a very similar perception and usage of SATD between industry and OSS. At the same time, some differences emerged, e.g., industrial developers are less prone than OSS developers to admit TD. This might be due to organizational policies and, above all, to the lack of a shared culture of not being afraid to share mistakes and information about imperfect code. Moreover, OSS developers care more about the need for API replacement than in industry, likely because of having more degrees of freedom.

The study results also highlight challenges developers are encountering upon admitting SATD. This promotes future research work in this area, aimed at supporting developers in the admittance of TD, providing better notification mechanisms for that, and also highlighting and discouraging practices related to removing TD documentation without properly addressing it.

References

Aghajani E, Nagy C, Vega-Márquez OL, Linares-Vásquez M, Moreno L, Bavota G, Lanza M (2019) Software documentation issues unveiled. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 1199–1210

Aghajani E, Nagy C, Linares-Vásquez M, Moreno L, Bavota G, Lanza M, Shepherd DC (2020) Software documentation: The practitioners' perspective. In: 2020 IEEE/ACM 42nd international conference on software engineering (ICSE). IEEE

Alexeeva Z, Perez-Palacin D, Mirandola R (2016) Design decision documentation: A literature overview. In: Tekinerdogan B, Zdun U, Babar MA (eds) Software Architecture - 10th european conference, ECSA 2016, November 28 - December 2, 2016, Proceedings, Lecture Notes in Computer Science, vol 9839, Denmark, pp 84–101

de Almeida RR, Kulesza U, Treude C, Feitosa DC, Lima AHG (2018) Aligning technical debt prioritization with business objectives: A multiple-case study. In: 2018 IEEE international conference on software maintenance and evolution, ICSME 2018, September 23-29, 2018, Spain, pp 655–664

de Almeida RR, Treude C, Kulesza U (2019) Tracy: A business-driven technical debt prioritization framework. In: 2019 IEEE international conference on software maintenance and evolution, ICSME 2019, September 29 - October 4, 2019, USA, pp 181–185

Alves NSR, Ribeiro LF, Caires V, Mendes TS, Spínola RO (2014) Towards an ontology of terms on technical debt. In: Sixth international workshop on managing technical debt, MTD ICSME 2014, September 30, 2014, Canada, pp 1–7

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: 2009 IEEE 31st international conference on software engineering, pp 298–308

Arnaoudova V, Eshkevari LM, Di Penta M, Oliveto R, Antoniol G, Gueheneuc YG (2014) Repent: Analyzing the nature of identifier renamings. IEEE Trans Softw Eng 40(5):502–532

Arnaoudova V, Di Penta M, Antoniol G (2016) Linguistic antipatterns: What they are and how developers perceive them. Empir Softw Eng 21(1):104–158

Avelino G, Passos L, Hora A, Valente MT (2016) A novel approach for estimating truck factors. In: 2016 IEEE 24th international conference on program comprehension (ICPC). IEEE, pp 1–10

Avelino G, Constantinou E, Valente MT, Serebrenik A (2019) On the abandonment and survival of open source projects: An empirical investigation. In: 2019 ACM/IEEE international symposium on empirical software engineering and measurement, ESEM 2019, September 19-20, 2019. IEEE, Brazil, pp 1–12

Baltes S, Ralph P (2020) Sampling in software engineering research: A critical review and guidelines. arXiv:200207764

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of the 13th international conference on mining software repositories, MSR 2016, May 14-22, 2016, USA, pp 315–326

Bellomo S, Nord RL, Ozkaya I, Popeck M (2016) Got technical debt?: surfacing elusive technical debt in issue trackers. In: Kim M, Robbes R, Bird C (eds) Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, May 14-22, 2016. ACM, USA, pp 327–338. https://doi.org/10.1145/2901739.2901754

Benjamini Y, Hochberg Y (1995) Controlling the false discovery rate: A practical and powerful approach to multiple testing. J R Statist Soc Ser B Methodol 57(1):289–300

Besker T, Martini A, Lokuge RE, Blincoe K, Bosch J (2018) Embracing technical debt, from a startup company perspective. In: 2018 IEEE international conference on software maintenance and evolution, ICSME 2018, September 23-29, 2018, Spain, pp 415–425

Brown N, Cai Y, Guo Y, Kazman R, Kim M, Kruchten P, Lim E, MacCormack A, Nord R, Ozkaya I et al (2010) Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP workshop on future of software engineering research. ACM

Celik A, Knaust A, Milicevic A, Gligoric M (2016) Build system with lazy retrieval for java projects. In: Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering, pp 643–654

Chen C, Zhang K, Itoh T (2012) Empirical evidence of tags supporting high-level awareness. In: Luo Y (ed) Cooperative design, visualization, and engineering - 9th international conference, CDVE 2012, September 2-5, 2012. Proceedings, Lecture Notes in Computer Science, vol 7467. Springer, Japan, pp 94–101

Cunningham W (1992) The WyCash portfolio management system. In: Addendum to the proceedings on object-oriented programming systems, languages, and applications. ACM

Di Nucci D, Palomba F, De Rosa G, Bavota G, Oliveto R, De Lucia A (2017) A developer centered bug prediction model. IEEE Trans Softw Eng 44(1):5–24

Ebert F, Castor F, Novielli N, Serebrenik A (2019) Confusion in code reviews: Reasons, impacts, and coping strategies. In: Wang X, Lo D, Shihab E (eds) 26th IEEE international conference on software analysis, evolution and reengineering, SANER 2019, February 24-27, 2019. IEEE, China, pp 49–60

Ernst NA, Bellomo S, Ozkaya I, Nord RL, Gorton I (2015) Measure it? manage it? ignore it? software practitioners and technical debt. In: Proceedings of the 2015 10th joint meeting on foundations of software engineering, ESEC/FSE 2015, August 30 - September 4, 2015, Italy, pp 50–60

Etco J (2017) todo: Automatically generate new issues. https://todo.jasonet.co/. Accessed 06 May 2020

Falessi D, Briand LC, Cantone G, Capilla R, Kruchten P (2013) The value of design rationale information. ACM Trans Softw Eng Methodol 22(3):21:1–21:32

Fisher RA (1922) On the interpretation of chi-square from contingency tables, and the calculation of p. J R Stat Soc 85(1):87–94

Fucci G, Zampetti F, Serebrenik A, Di Penta M (2020) Who (self) admits technical debt. In: 2020 IEEE international conference on software maintenance and evolution, ICSME 2020. IEEE

Fucci G, Cassee N, Zampetti F, Novielli N, Serebrenik A, Di Penta M (2021) Waiting around or job half-done? sentiment in self-admitted technical debt. In: International conference on mining software repositories. IEEE Computer Society, United States

Grissom RJ, Kim JJ (2005) Effect sizes for research: A broad practical approach, 2nd edn. Lawrence Earlbaum Associates, Mahwah

Guzzi A (2012) Documenting and sharing knowledge about code. In: 2012 34th international conference on software engineering (ICSE). IEEE, pp 1535–1538

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

Iammarino M, Zampetti F, Aversano L, Di Penta M (2019) Self-admitted technical debt removal and refactoring actions: Co-occurrence or more? In: 2019 IEEE international conference on software maintenance and evolution, ICSME 2019, September 29 - October 4, 2019. IEEE, USA, pp 186–190

Jiang S, Armaly A, McMillan C (2017) Automatically generating commit messages from diffs using neural machine translation. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, October 30 - November 03, 2017, USA, pp 135–146

Kruchten P, Nord RL, Ozkaya I, Falessi D (2013) Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt. ACM SIGSOFT Softw Eng Notes

Kruskal WH, Wallis WA (1952) Use of ranks in one-criterion variance analysis. J Am Stat Assoc 47(260):583–621. http://www.jstor.org/stable/2280779

Li Z, Avgeriou P, Liang P (2015) A systematic mapping study on technical debt and its management. J Syst Softw 101:193–220

Lim E, Taksande N, Seaman C (2012) A balancing act: what software practitioners have to say about technical debt. IEEE Softw

Lin B, Zampetti F, Bavota G, Di Penta M, Lanza M (2019) Pattern-based mining of opinions in q&a websites. In: Proceedings of the 41st international conference on software engineering, ICSE 2019, May 25-31, 2019, Canada, pp 548–559

Maipradit R, Treude C, Hata H, Matsumoto K (2020) Wait for it: identifying "on-hold" self-admitted technical debt. Empir Softw Eng :1–29

Mensah S, Keung J, Svajlenko J, Bennin KE, Mi Q (2018) On the value of a prioritization scheme for resolving self-admitted technical debt. J Syst Softw

Moreno L, Bavota G, Di Penta M, Oliveto R, Marcus A, Canfora G (2017) ARENA: an approach for the automated generation of release notes. IEEE Trans Software Eng 43(2):106–127

Newcombe RG (1998) Two-sided confidence intervals for the single proportion: comparison of seven methods. Stat Med 17(8):857–872

Padioleau Y, Tan L, Zhou Y (2009) Listening to programmers - taxonomies and characteristics of comments in operating system code. In: Proceedings 31st international conference on software engineering, ICSE 2009, May 16-24, 2009. IEEE, Canada, pp 331–341

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: 30th IEEE international conference on software maintenance and evolution, September 29 - October 3, 2014, Canada, pp 91–100

Rantala L, Mäntylä M, Lo D (2020) Prevalence, contents and automatic detection of kl-satd. arXiv:200805159

Rastkar S, Murphy GC, Murray G (2014) Automatic summarization of bug reports. IEEE Trans Softw Eng 40(4):366–380

Ren X, Xing Z, Xia X, Lo D, Wang X, Grundy J (2019) Neural network-based detection of self-admitted technical debt: From performance to explainability. ACM Trans Softw Eng Methodol 28(3):15

da S Maldonado E, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: 7th IEEE international workshop on managing technical debt, MTD 2015, October 2, 2015, Germany, pp 9–15

da S Maldonado E, Abdalkareem R, Shihab E, Serebrenik A (2017a) An empirical study on the removal of self-admitted technical debt. In: 2017 IEEE international conference on software maintenance and evolution, ICSME 2017, September 17-22, 2017, China, pp 238–248

da S Maldonado E, Shihab E, Tsantalis N (2017b) Using natural language processing to automatically detect self-admitted technical debt. IEEE Trans Software Eng 43(11):1044–1062

Seaman C, Guo Y (2011) Measuring and monitoring technical debt. Adv Comput

Spencer D (2009) Card sorting: Designing usable categories. Rosenfeld Media

Steele CM, Aronson J (1995) Stereotype threat and the intellectual test performance of african americans. J Pers Soc Psychol 69(5):797–811

Storey MA, Cheng LT, Bull I, Rigby P (2006) Shared waypoints and social tagging to support collaboration in software development. In: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, pp 195–198

Storey MA, Ryall J, Bull RI, Myers D, Singer J (2008) TODO or to Bug: Exploring how task annotations play a role in the work practices of software developers. In: Proceedings of the 30th international conference on software engineering, association for computing machinery, USA, pp 251–260

Storey MA, Ryall J, Singer J, Myers D, Cheng L, Muller MJ (2009) How software developers use tagging to support reminding and refinding. IEEE Trans Softw Eng 35(4):470–483

Tan SH, Marinov D, Tan L, Leavens GT (2012) @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In: Fifth IEEE international conference on software testing, verification and validation, ICST 2012, April 17-21, 2012, Canada, pp 260–269

Torchiano M, Ricca F, Marchetto A (2011) Is my project's truck factor low? theoretical and empirical considerations about the truck factor threshold. In: Proceedings of the 2nd international workshop on emerging trends in software metrics, pp 12–18

Uddin G, Khomh F (2017) Opiner: an opinion search and summarization engine for APIs. In: Proceedings of the 32nd IEEE/ACM international conference on automated software engineering, ASE 2017, October 30 - November 03, 2017, USA, pp 978–983

Vassallo C, Zampetti F, Romano D, Beller M, Panichella A, Di Penta M, Zaidman A (2016) Continuous delivery practices in a large financial organization. In: 2016 IEEE international conference on software maintenance and evolution, ICSME 2016 October 2-7, 2016. IEEE Computer Society, USA, pp 519–528

Vassallo C, Proksch S, Gall HC, Di Penta M (2019) Automated reporting of anti-patterns and decay in continuous integration. In: 2019 IEEE/ACM 41st international conference on software engineering (ICSE). IEEE, pp 105–115

Wehaibi S, Shihab E, Guerrouj L (2016) Examining the impact of self-admitted technical debt on software quality. In: IEEE 23rd international conference on software analysis, evolution, and reengineering, SANER 2016, March 14-18, 2016. IEEE Computer Society, Japan, pp 179–188

Wei L, Liu Y, Cheung SC (2017) Oasis: prioritizing static analysis warnings for android apps based on app user reviews. In: Proceedings of the 2017 11th joint meeting on foundations of software engineering, pp 672–682

Wen F, Nagy C, Bavota G, Lanza M (2019) A large-scale empirical study on code-comment inconsistencies. In: 2019 IEEE/ACM 27th international conference on program comprehension (ICPC). IEEE, pp 53–64

Wilcoxon F (1945) Individual comparisons by ranking methods. Biom Bull 1(6):80–83

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

Zampetti F, Noiseux C, Antoniol G, Khomh F, Di Penta M (2017) Recommending when design technical debt should be self-admitted. In: 2017 IEEE international conference on software maintenance and evolution, ICSME 2017, September 17-22, 2017, China, pp 216–226

Zampetti F, Serebrenik A, Di Penta M (2018) Was self-admitted technical debt removal a real removal?: an in-depth perspective. In: Proceedings of the 15th international conference on mining software repositories, MSR 2018, May 28-29, 2018, Sweden, pp 526–536

Zampetti F, Serebrenik A, Di PentaM (2020) Automatically learning patterns for self-admitted technical debt removal. In: 2020 IEEE 27th international conference on software analysis evolution and reengineering (SANER), pp 355–366

Zampetti F, Fucci G, Serebrenik A, Di Penta M (2021) Dataset of the paper "self-admitted technical debt practices: A comparison between industry and open- source". https://doi.org/10.5281/zenodo.5076096

Zazworka N, Shaw MA, Shull F, Seaman CB (2011) Investigating the impact of design debt on software quality. In: Proceedings of the 2nd workshop on managing technical debt, MTD 2011, May 23, 2011, USA, pp 17–23

Zimmermann T (2016) Card-sorting: From text to themes. In: Menzies T, Williams L, Zimmermann T (eds) Perspectives on data science for software engineering. Morgan Kaufmann, Boston, pp 137–141

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Fiorella Zampetti is a Postdoctoral Researcher at University of Sannio, Italy. She received her Ph.D. degree in Information Technologies for Engineering from the University of Sannio (Italy) in 2019. Her current research is focused on DevOps for Cyber-Physical Systems (CPSs), software maintenance and evolution, and mining software repositories. She has served as a program committee member of international conferences in the software engineering field, and she serves as a reviewer for Software Engineering journals including TOSEM, TSE, EMSE, and JSEP. She is part of the organizing committee of the SBST tool competition.

Gianmarco Fucci is a PhD student at the University of Sannio under the supervision of Prof. Massimiliano Di Penta and Research Technician at the Ugo Bordoni Foundation in Rome. He was born in Benevento on March 14, 1994. He received his Bachelor's Degree in Computer Engineering in July 2016 and his Master's Degree in December 2018, both from the University of Sannio. His current fields of interest are Software Engineering and Cyber-security.

Alexander Serebrenik is a Full Professor of Social Software Engineering at Eindhoven University of Technology, The Netherlands. His research goal is to facilitate evolution of software by taking into account social aspects of software development. He has co-authored a book Evolving Software Systems (Springer Verlag, 2014) and circa 200 scientific papers and articles. He has won several distinguished paper and distinguished review awards.

Massimiliano Di Penta is a full professor at the University of Sannio, Italy. His research interests include software maintenance and evolution, mining software repositories, empirical software engineering, search-based software engineering, and software testing. He is an author of over 300 papers that appeared in international journals, conferences, and workshops. He has received several awards for research and service, including four ACM SIGSOFT Distinguished paper awards. He serves and has served in the organizing and program committees of more than 100 conferences, including ICSE, FSE, ASE, and ICSME. He is co-editor in chief of the Journal of Software: Evolution and Processes edited by Wiley, editorial board member of ACM Transactions on Software Engineering and Methodology and Empirical Software Engineering Journal edited by Springer, and has served the editorial board of the IEEE Transactions on Software Engineering.