2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR) | 978-1-7281-8710-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/MSR52588.2021.00052

Waiting around or job half-done? Sentiment in self-admitted technical debt

Gianmarco Fucci*, Nathan Cassee†, Fiorella Zampetti*, Nicole Novielli‡, Alexander Serebrenik†, Massimiliano Di Penta*
*University of Sannio, Italy {gianmarco.fucci,fiorella.zampetti,dipenta}@unisannio.it
†Eindhoven University of Technology, The Netherlands {n.w.cassee,a.serebrenik}@tue.nl
‡University of Bari, Italy nicole.novielli@uniba.it

***Abstract*—Self-Admitted Technical Debt (SATD) represents the admission, made through source code comments or other channels, of portions of a program being poorly implemented, containing provisional solutions or, in general, simply being not ready yet. To better understand developers' habits in SATD annotation, and possibly support their exploitation in tool support, this paper provides an in-depth analysis of the content provided in SATD comments, and the expressed sentiment. We manually inspect and classify 1038 instances from an existing dataset, grouping them along a taxonomy composed of 41 categories (of which 9 top-level ones), identifying their sentiment, and the presence of external references such as author names or issue IDs. Results of our study indicate that (i) the SATD content is crosscutting along life-cycle dimensions identified in previous work, (ii) comments related to functional problems or on-hold SATD are generally more negative than poor implementation choices or partially implemented functionality, and (iii) despite observations from previous literature, only a minority of SATD comments leverage external references.**

***Index Terms*—Self-Admitted technical debt; sentiment analysis; empirical study**

I. Introduction

Self-admitted technical debt (SATD) [1] are source code comments indicating that the corresponding source code is (temporarily) inadequate, e.g., because the implementation is incomplete, buggy, or smelly. The identification of SATD [2], [3], as well as its introduction or removal have attracted significant attention of the research community [4]–[6].

To understand what kind of TD need to be addressed, previous work also categorized SATD comments. Categorizations of SATD proposed so far are based on the phases of the software development process [7], [8] and as such (a) miss the opportunity to identify concerns transcending the boundaries of individual phases such as waiting for other components to be ready, and (b) are somewhat broad because the SATD content still lacks an in-depth classification. Moreover, while SATD might manifest at one phase of the software development phase, resolving it might require activities typically associated with another phase. For instance, the following SATD comment can be expected to manifest during testing but its resolution requires a bug to be fixed, a typical implementation activity: "doesn't work: Depending on the compression engine used, compressed bytes may differ. False errors would be reported. `assertTrue(''File content mismatch'', FILE_UTILS.contentEquals(...));`." Hence, while the existing classifications contribute to the understanding of the SATD phenomenon, designing recommenders supporting SATD resolution calls for a different categorization of the highlighted problems. By providing a more fine-grained classification of the problems experienced by contributors we expect that more actionable insights can be obtained from SATD. Thus, we ask the following research question:

> **RQ**$_1$: *What kind of problems do SATD annotations describe?*

To address **RQ**$_1$, we first take 1038 SATD comments sampled from the dataset of Maldonado et al. [2] and perform a fine-grained classification. We classify SATD comments from the point of view of *their textual content*, as opposed to a *software development life-cycle*, as it was done in previous work [7], [8]. We create our taxonomy using a bottom-up strategy (i.e., what do SATD comments mention?) rather than top-down (i.e., how do SATD comments map onto a software development life-cycle?). This leads us towards a taxonomy featuring 9 top-level categories specialized into 32 sub-categories. The taxonomy spotlights categories that are, on the one hand, crosscutting to the life-cycle and, on the other hand, more related to the reasons why SATD was admitted and to the goal developers are trying to achieve.

As projects might have hundreds of SATD comments [5], resolving them requires prioritization. Specifically, previous work by Pletea et al. [9] found that issues and pull requests comments reporting security concerns convey more negative sentiment than others. Furthermore, timely detection of negative emotions, such as frustration, might be leveraged to identify and support developers experiencing difficulties [10], thus preventing burnout and loss of productivity [11]. Similarly to previous research [10], [12], [13] we conjecture that the presence of a negative sentiment as a proxy to the comment importance as perceived by the developers themselves. Additionally, we expect certain kind of problems discussed in the SATD to be be associated with a stronger negative sentiment than others. Therefore we ask:

> **RQ**$_2$: *How is the sentiment polarity distributed across different kinds of SATD annotations?*

To address **RQ**$_2$ we label sentiment polarity on the 1038 SATD comments used for addressing RQ$_1$. We label as non-

978-1-7281-8710-5/21/$31.00 ©2021 IEEE
DOI 10.1109/MSR52588.2021.00052

negative all comments merely stating the problem or suggesting an improvement, e.g., "TO DO : delete the file if it is not a valid file,". Conversely, we considered as negative all comments expressing a negative attitude, e.g., "TODO : YUCK!!! fix after HHH-1907 is complete".

Surprisingly we observe that only 30% of the SATD comments convey negative sentiment. To get a more refined understanding of the way the sentiment is expressed in SATD comments, we analyzed the extent to which different categories of our taxonomy, as well as different categories of Maldonado et al. [2], exhibit a different distribution of sentiment polarity: we observe that the percentage of negative comments associated with functional issues (49%) and waiting (46%) is much higher than associated with an incomplete implementation (13%).

Finally, SATD annotations are closely related to the task annotations studied by Storey et al. [14], as the task annotations use similar keywords as the SATD comments considered in this study. Storey et al. surveyed 70 developers to understand what additional details they tend to record when adding task annotations to the source code. Almost two-thirds of the respondents declared that they add references to other classes, methods, plug-ins, or modules; more than half of the respondents include their name or initials in the source code annotations, 44% include references to bugs, 30% to the URLs, 19% record the date, 10% record "memorable keywords" and merely 13% do not add additional details. Using the dataset collected to answer $\mathbf{RQ}_1$ and $\mathbf{RQ}_2$ we conduct a conceptual replication [15] of this study and ask:

> $\mathbf{RQ}_3$: *To what extent do SATD annotations belonging to different categories contain additional details?*

We focus on references to class and method names, authors' names and initials, comment dates, as well as pointers to issues and external documents/URLs. However, we automatically or manually mine them rather than asking for developers' perception. Our results indicate that, although some types of references are perceived as extremely important by participants of Storey et al. [14] study, they infrequently appear in SATD comments, however, developers tend to mention more frequently classes and/or methods, probably for improving the overall traceability.

The full dataset, and files used during the annotation, are publicly available.[1]

II. Study Design

The *goal* of the study is to analyze SATD comments, to categorize their content and, in general, understand the way developers communicate the presence of SATD in source code.

A. Study Context

We start from a curated dataset of SATD comments by Maldonado et al. [2], consisting of 4071 SATD comments belonging to 10 different open source Java projects. These comments were classified by Maldonado et al. [2] into five categories (Defect, Design, Documentation, Implementation, and Test); Implementation features also Requirement debt from the original taxonomy of Maldonado and Shihab [7].

[1] https://figshare.com/s/0b83bc75dbc9ea99f2f6

TABLE I
NUMBER OF SATD COMMENTS IN THE ORIGINAL DATASET AND IN THE SAMPLED ONES.

SATD Type	Initial Dataset	Without Duplication	Sampled
Defect	472	350	116
Design	2703	2260	657
Documentation	54	49	39
Implementation	757	550	183
Test	85	80	43
TOTAL	4071	3289	1038

First, we remove 782 duplicated comments (i.e., comments having the same content but attached to different source code elements) since our focus is on the content. After the removal, we manually analyze a statistically significant sample of 1038 SATD comments (confidence interval of $\pm 3.33\%$ for a confidence level of 99%). As reported in Table I, our sample has the same percentage of SATD comments types in the initial dataset, guaranteeing that each SATD type is well represented in our study. For instance, our dataset without duplication counts 350 SATD belonging to DEFECT, i.e., $\simeq 11\%$ over the total number of SATD comments (3289), and in our sample, we have manually analyzed 116 SATD comments in the same category that accounts for 11% of the total number of SATD comments being analyzed.

B. Addressing RQ_1: SATD content coding

To derive a SATD contents' taxonomy, the manual analysis has been done following a card-sorting procedure, and specifically a cooperative (multiple annotators) open card-sorting (no predefined categories) [16].

In a first round, two of the authors independently created labels for 108 SATD comments randomly chosen from the dataset without duplication in proportion to each SATD type. After completing, the two annotators discussed their labels, i.e., also resolving inconsistencies and redundancies, and grouped the tags into a hierarchy. After that, two different authors reviewed the initial set of created labels, in turn suggesting improvements. After the first round, the authors end up with a taxonomy made up of 11 high-level categories specialized into a total of 26 sub-categories.

In a second round, two authors used the first version of the taxonomy to label a different set of 115 SATD comments randomly picked from the dataset without duplicated instances and, again in proportion to each SATD type. More specifically, while reading a SATD comment content the annotator could choose to reuse an existing label or to add a new one. Upon completion, the two annotators solved inconsistencies and evaluated the introduction of newly added labels. The updated version of the taxonomy has been sent to two different authors, that after some improvements ended up with a taxonomy accounting for ten high-level categories specialized into a total of 28 sub-categories. More specifically, two high-level categories have been used as specialization of other categories and one has been added (see details in the online dataset).

In a third round, using the same process, the authors manually analyzed 114 SATD comments. As a result, they obtained a new modified version of the taxonomy made up of 11 high-level categories, of which two are newly introduced ones and one became a sub-category. The high-level categories were properly specialized into a total of 36 sub-categories, five of which were not reported in the previous version.

This final version of the taxonomy has been used to label the remaining 701 comments that were randomly assigned to four authors, such that each SATD comment was independently analyzed by two of them. Also in this case, the annotators could either use the existing labels or create a new one if no one fitted a specific comment. As it happens in teamwork card-sorting [16], newly introduced labels (groups) became immediately available also for other annotators. Upon completion, the annotators discussed their classifications resolving inconsistencies and revised the taxonomy. In this round, the authors did not introduce any new high-level category while using two of them to specialize existing ones, even if there is the introduction of two new sub-categories. In summary, since in our last round no new high-level categories are introduced, the identified taxonomy is general enough. However, this does not exclude that, in the future, further contents could emerge and be therefore included in the taxonomy.

To address RQ_1, we present our final version of the taxonomy, explaining, with some examples, the identified categories, and also highlighting the number of comments found for each category.

C. Addressing RQ_2: Sentiment annotation

To address RQ_2, all 1038 comments have been manually annotated with sentiment polarity. By definition, SATD describes an undesirable situation, so we do not expect to observe many positive comments and opt to classify sentiment as either *negative* or *non-negative*, where the latter category includes both positive and neutral comments. Comments conveying both positive and negative sentiment are labeled as *mixed*. We label as negative comments containing expressions that clearly communicate negative sentiment, e.g., emotions or negative opinions about the underlying code, beyond the negativity inherent in problem reporting as in SATD comments.

Determining sentiment for a text is a subjective task, i.e., the labels given by individuals depend on their cultural background, upbringing, and interpretation of the comment [17]. As such, following clear annotation guidelines is recommended for enabling reliable annotation [18]. To ensure robust and consistent labeling, we defined a set of annotation guidelines by conducting a pilot labeling study. We randomly sampled 32 comments from the 1038 comments and asked each author to label them individually, based on their subjective perception of each comment polarity. Then, we jointly discussed disagreements in a plenary session, resolving conflicts and addressing ambiguities in the definition of negative sentiment. Based on the results of our discussion, we drafted our coding guidelines to be used for the labeling study as follows:

- *negative*: If the comment expresses negative sentiment about the underlying source-code (e.g., "this method is a nightmare");
- *non-negative*: If the comment expresses either positive or no sentiment about the code referenced in the comment (e.g., "TODO: Why is this a special case?");
- *mixed*: If the comment expresses both positive and negative sentiment (e.g., "This is a fairly specific hack for empty string, but it does the job").

Before proceeding with manual labeling, we explored the possibility of automating sentiment analysis of SATD. To this aim, we evaluated the accuracy of three publicly available sentiment analysis tools that have been specifically tuned for the software engineering domain, i.e., SentiStrength-SE [19], Senti4SD [20], and SentiCR [21] on the 32 SATD comments manually labeled. We apply the tools "off-the-shelf", i.e., without further tuning or training [22]. Looking at the agreement between manual labels and the tools' predictions as recommended by Novielli et al. [18], [23], we found that Senti4SD has the highest F-1 score (0.69), lower than the one reported by the authors of the tool (0.87) [20]. By manually looking at disagreements we found that some negative comments were missed by the tool due to the presence of lexicon which is specific to SATD comments. For example, "FIXME: Big fat hack here, because scope names are expected to be interned strings by the parser" is labeled as negative by the human judges but classified as neutral by Senti4SD. We conclude that the operationalization of sentiment by tools does not align with our operationalization of sentiment in SATD, and decide to manually label all remaining SATD comments in our dataset.

We divided the comments in our sample, excluding the ones already labeled in our pilot study (1006), over the six authors of the paper, such that each author labeled an equal number of comments per SATD category, and each comment was labeled by at least two authors, to mitigate the presence of any biases between authors and over SATD categories. Finally, to ensure reliability and consistency of our labeled dataset, we resolved all disagreements in plenary sessions involving all raters. The agreement between the raters for the sentiment labeling is moderate, with a Krippendorf's α of 0.455 [24], which is in line with agreement reported by previous studies on developers' sentiment annotation in short comments from software development platforms [25]. Lastly, to understand for what SATD categories negative sentiment is more likely to occur, and how this differs over the SATD categories of the taxonomy constructed in RQ_1, we use a pairwise proportion test [26] with Benjamini-Hochberg p-value adjustment [27].

D. Addressing RQ_3: Identifying Additional Details in SATD

While looking at task annotations, Storey et al. [14] have noticed that when adding task annotations developers frequently include: (i) references to another class, method, plug-in, or module, (ii) developers' names or initials, (iii) references to bugs, (iv) URLs, (v) dates, and (vi) "memorable keywords".

To understand how often these additional details occur in SATD we use a combination of manual labeling and automated

detection to extract fields (i) through (vi) from the 1038 SATD comments. Due to the heterogeneity (as well as our unfamiliarity) with the practices of the projects that make up the dataset, and considering that in SATD comments keywords are mainly related to tags, e.g., TODO, FIXME, XXX etc. we have chosen to not identify "memorable keywords".

Firstly, we identify the following fields automatically:

- for class names, we search for all possible class names of a project, obtained from its git repository (all file versions from all branches), onto comments, using a case insensitive, word boundary match, and for methods references we use a simple regular expression ("$\backslash\backslash w + \backslash\backslash($");
- for bug references, we use the Fischer et al. approach [28], e.g., matching JIRA-style references (e.g., "jruby-1234") or GitHub-style reference (e.g., "#1234");
- for URLs we match the following two regular expressions onto the SATD comments, i.e., `http://` and `https://`.

We initially automated the identification of author names (based on names from the versioning systems), but this turned out to be non-trivial and imprecise, therefore we decided to check them manually, following a process similar to that outlined in Section II-C. Also, while we initially detected issue-IDs and dates (in this case matching various formats as "12 Jan 2002", or "20020112") automatically, we double-checked them manually because of the presence of several formats.

Based on a manual inspection of the dataset, we combine the results of the automatic detection with the manual labeling. Specifically:

(i) automatically identified for class/method names and URLs we use the automated detection;
(ii) for developers names/initials and dates we rely on the manual labeling;
(iii) for bugs we combine the manual analysis with the automatic detection.

We report the occurrences of each field per macro-category of the taxonomy, and evaluate how the perceptions of developers, as found by Storey et al. [14], compare to the occurrences of these fields in the 1038 SATD comments.

III. Study Results

This section reports and discusses the study results, addressing the research questions formulated in the introduction.

A. What kind of problems do SATD annotations describe?

Fig. 1 depicts the taxonomy of SATD comments' content, obtained as described in Section II-B. Below we describe our taxonomy, and then explain why SATD comments from source code do not naturally fit the categories from Maldonado et al.

The small red boxes of Fig. 1 indicate the number of SATD comments (out of 1038) belonging to each category. Table II reports the distribution of our taxonomy top-level categories. For some comments we were able to identify the higher-level category but not any sub-categories: e.g., "TODO: implement the entity for the annotation" in JFREECHART is an indication that the functionality is only partially implemented but does not contain any other information aimed at justifying why that happened.

Although data originated from a curated dataset [4], we still found 40 instances that, according to our manual analysis, were not related to SATD. For example, "Required otherwise it gets too wide" in SQL explains the design decision without indicating that it is suboptimal in any sense. We classified them as false positives and excluded from Fig. 1.

While (not surprisingly) most SATD comments highlight poor implementation choices (429 over 1038) mainly related to maintainability issues, as well as partially/not implemented functionality (229), we notice that functional issues (135) are not so frequent in our sample. Furthermore, we found 89 SATD comments classified as "Wait", meaning that a developer cannot improve the code or complete a functionality since they are waiting for a different event that has to occur in the same project or in a third-party component (e.g., "this is the temporary solution for issue 1011" in JFREECHART). As also reported in previous work [7], [8], developers tend to admit TD also in artifacts that are different from the production code: indeed, we found 54 SATD comments dealing with issues in the documentation, and 36—with the test code. Finally, we found 21 SATD comments describing misalignment between requirements and design or implementation, as well as problems with deployment (2) and SATD comments that are left in the code while not describing a TD anymore (3).

Next, we elaborate on each of the nine high-level categories of our taxonomy.

Poor implementation choice. This category includes (i) maintainability issues, (ii) poor implementation solutions, (iii) asking for code review, i.e., the developer is not sure of the actual design, (iv) performance issues, (v) poor API usages, i.e., reliance on a third-party component without actually understanding the proper way to use it, (vi) lack of intention to improve the code despite the awareness that it is not in the right shape, and (vii) usability issues.

Maintainability issues constitute the category with the highest number of samples, not merely within "Poor implementation choices" but overall, covering 20% of the comments. Unsurprisingly, many maintainability issues require a refactoring activity such as a better distribution of responsibilities among software components (e.g., "TODO: We should have all the information that is required in the NotationSettings object" in ARGOUML), or proper reuse of features (e.g., "TODO: Reuse the offender List" in ARGOUML).

Furthermore, we found 79 SATD comments reporting that the implemented solution has to be improved, e.g., "EATM This might be better written as a single loop for the EObject case" in EMF highlighting the need for simplifying the actual implementation removing a control structure. In other cases, the authors criticize the implementation choices and ask for a code review, e.g., "FIXME: Is "No Namespace is Empty Namespace" really OK?" in APACHE-ANT or "TODO: this assumes ranges are sorted. Is this true?" in ARGOUML. This

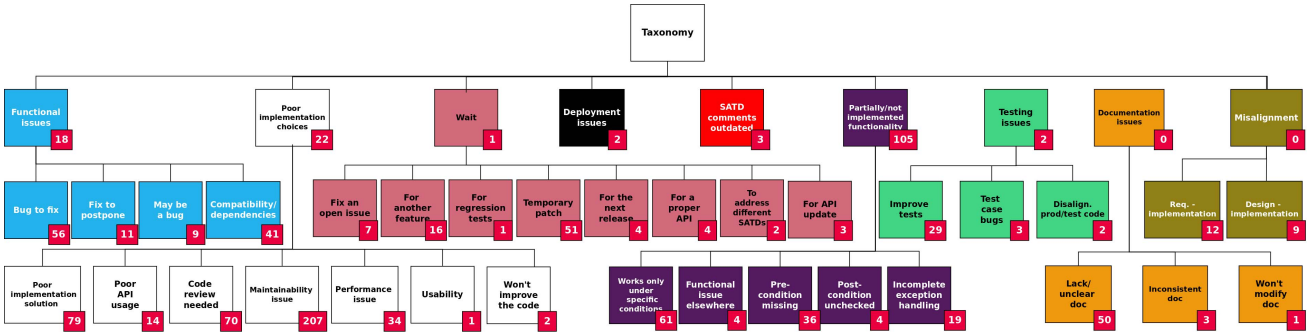

Fig. 1. Discussions contents in SATD comments

TABLE II
DISTRIBUTION OF OUR TAXONOMY TOP-LEVEL CATEGORIES AND HOW THEY MAP ONTO MALDONADO ET AL. [2] CATEGORIES.

Macro-category	Defect	Design	Doc.	Impl.	Test	Total
Functional issues	48	68	0	18	1	135
Poor implementation choices	22	361	2	43	1	429
Wait	6	76	0	6	1	89
Deployment issues	1	0	0	1	0	2
SATD comments outdated	2	1	0	0	0	3
Partially implemented	27	94	5	100	3	229
Testing issues	0	1	0	0	35	36
Documentation issues	0	19	30	4	1	54
Misalignment	1	13	2	5	0	21
False positive	9	24	0	6	1	40
TOTAL	116	657	39	183	43	1038

result confirms the findings by Ebert et al. [29] who highlight that 8% of questions during code reviews express attitudes and emotions. Specifically, their manual coding shows that developers express doubts through criticisms ($\simeq 5\%$) inducing critical reflection in the interlocutor.

Moreover, concerns related to the use of APIs and performance are reflected in the SATD comments: e.g., "FIXME: don't use RubyIO for this" in JRUBY alerts developers to replace the existing API for a specific task, while "TODO replace repeated `substr()` above and below with more efficient method" in JMETER indicates performance issues.

Partially/Not implemented functionality groups the SATD comments reporting that a feature is not ready yet. While, on the one hand, we found many cases (105) in which the SATD comment simply reports that the implementation is missing without adding any further details, on the other hand, we found comments indicating what is specifically missing from the implementation: e.g., a precondition ("TODO: delete the file if it is not a valid file" in ANT), or a postcondition check ("FIXME: Make bodyNode non-null in parser" in JRUBY).

We found comments clarifying that the feature works only under specific conditions (61) as "If c2 is empty, then we're done. If c2 has more than one element, then the model is crappy, but we'll just use one of them anyway" in ARGOUML.

Some comments indicate that the implementation is absent due to problems elsewhere: e.g., "Predecessors used to be not implemented, because it caused some problems that I've not found an easy way to handle yet. The specific problem is that the notation currently is ambiguous on second message after a thread split." in COLUMBA.

Functional issue includes all cases directly or indirectly related to the presence of a bug in the system and constitutes the third-largest category of SATD comments. Unsurprisingly, most of them (56) highlight the presence of a bug that should be fixed immediately: e.g., "FIXME: If `NativeException` is expected to be used from Ruby code, it should provide a real allocator to be used. Otherwise `Class.new` will fail, as will marshaling. JRUBY-415" in JRUBY. 11 SATD comments indicate misbehavior that is acceptable even though a better solution has to be found: e.g., "this will generate false positives but we can live with that" in ANT.

The most interesting sub-category groups compatibility and dependency issues that are also not very easy to address. For instance, we found comments indicating that the code is not able to work properly in specific environments, e.g., "`waitFor()` hangs on some Java implementations" in JEDIT, or cases where the actual implementation inherits a bug from an external API being used, e.g., "Workaround for JDK bug 4071281 [...] in JDK 1.2" in JEDIT.

Wait includes all SATD comments in which the author reports that the code has to be improved and/or completed once a different event occurs. In many cases (51) the comments report that the code is a temporary patch that needs to be removed later on, e.g., "TODO: temporary initial step towards HHH-1907" in HIBERNATE. Furthermore, 16 comments state that the code is not in the right shape since it requires a different feature to be ready first, e.g., "todo : remove this once `ComponentMetamodel` is complete and merged" in HIBERNATE. An interesting phenomenon related to waiting is an SATD comment requiring other SATD comments to be fixed, e.g., "TODO: simply remove this override if we fix the above todos" in HIBERNATE. We found four comments in which developers need to wait for a proper API to be found, e.g., "This really should be `Long.decode`, but there isn't one. As a result, hex and octal literals ending in 'l' or 'L' don't work." in JEDIT. Recently Maipradit et al. [30] have looked at "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, that maps onto our "Wait" category. Our results confirm what found by Maipradit et al. [30], since around 8% of the SATD comments contains a waiting condition, however, our taxonomy enlarges the set of possible events a developer is waiting for, indeed Maipradit et al. [30] only considered bugs to be fixed, or new releases/versions of libraries.

Documentation issues (54 over 1038). Many cases are related to the need for documenting a specific method/class such as "FIXME This function needs documentation" in COLUMBA. However, we also found three cases describing inconsistencies in the related documentation, e.g., "UML 1.4 spec is ambiguous - English says no Association or Generalization, but OCL only includes Association" in ARGOUML, and one case in which the author is reporting that the documentation cannot be modified even if it is required to modify it, i.e., "TODO: Currently a no-op, doc is read only" in ARGOUML.

Testing issues. Multiple SATD comments refer to test code, including (i) untested features, e.g., "TODO add tests to check for: - name clash - long option abbreviations/" in JMETER, (ii) bugs in the current test suite, e.g., "this is the wrong test if the remote OS is OpenVMS, but there doesn't seem to be a way to detect it" in ANT, or (iii) misalignment of the test code with the production code, e.g., "TODO: [...] An added test of `isAModel(obj)` or `isAProfile(obj)` would clarify what is going on here" in ARGOUML.

Misalignment groups the SATD comments in which the authors report that there is a mismatch between (i) requirements and implementation (12) such as "TODO: The Quickguide also mentions [...] Why are these gone?" in ARGOUML, in which the author asks whether the current implementation deviates from what reported in the specification, or (ii) design and implementation (9) such as "TODO: This shouldn't be public. Components desiring to inform the Explorer of changes should send events" in ARGOUML, clearly stating that there is a deviation from what reported in the design document.

We also found three instances belonging to **outdated SATD comments** in which the SATD comment no longer reflects the source code evolution e.g., "todo: is this comment still relevant ??" in ANT. This category generally belongs to the problem of comments being outdated with respect to source code. For simple cases, especially related to comments explaining statements' behavior, detection approaches have been proposed [31] and empirical studies have been carried out. As regards SATD, it is still possible that in many circumstances SATD comments remain in the system even after the mentioned problem has been addressed.

We found two SATD comments reporting **Deployment issues**. One comment in ARGOUML states: "As a future enhancement to this task, we may determine the name of the EJB JAR file using this display-name, but this has not be implemented yet.". The latter highlights the need for improvements to the overall deployment phase while constructing the application jar. The other comment in ANT states "the generated classes must not be added in the generic JAR! is that buggy on old JOnAS (2.4)", meaning that there is a problem while selecting the components to involve in the jar.

To understand the difference between our categories and those by Maldonado et al., Table II shows of SATD comments belonging to different categories of their taxonomy are mapped to our top-level ones. The categories of Maldonado et al. cut across several of our categories. For instance, although 41% (48 items) of SATD comments in the "Defect" category are mapped onto our "Functional issues", the remaining SATD comments are scattered onto the "Partially/not implemented functionality" and "Poor implementation choices". As an example of the former, the comment stating: "TODO: we didn't check the height yet" in JFREECHART originally considered as a defect SATD has been categorized as a "Partially/not implemented functionality" since the comment body has nothing reporting the presence of a bug in the system due to the lack of a pre-condition check. As regards the latter, instead, "TODO: This method doesn't appear to be used." in JMETER mostly highlights possible maintainability issues, therefore it has been categorized as a "Poor implementation choice".

Similarly, while "Design" SATD comments mainly belong to our "Poor Implementation Choices" category, some refer to waiting, e.g., "Remember to change this when the class changes" in JMETER, partially implemented functionality, e.g., "TODO: complete this" in JFREECHART or functional issues, e.g., "TODO - is this the correct default?" in JMETER.

The "Implementation" SATD comments were originally labeled as "Requirement debt" by Maldonado and Shihab [7] and then renamed in their follow-up dataset. While unsurprisingly almost half of them belong to our "Partially/Not implemented Functionality" (which is indeed requirement debt, because the requirement has not been fully implemented), 43 cases are related to tox poor implementation choices, hence not related to requirements, and better fitting our specific category. For example, there are comments in ARGOUML asking for code review, e.g., "TODO: Why is this disabled always?", or pointing out the presence of maintainability issues, e.g., "TODO: Reuse the offender List."

The categories of our taxonomy having a good fit with the ones of Maldonado et al. are "Documentation issues" and "Testing Issues". Still, in JMETER we found a documentation debt, e.g., "TODO Can't see anything in SPEC", we categorized as "Misalignment" since it relates to a discrepancy between specification and implementation, and either of the two can be wrong. As already reported in the introduction, the Maldonado et al. taxonomy classifies as "testing issues" cases of failed assertions, which we classified as "Functional issues" instead.

RQ_1 Summary: We categorized the sample of 1038 SATD comments into nine top-level categories, separating functional errors from partially implemented functionality and poor implementation choices. We also considered on-hold TD ("Wait") as a specific category with 89 instances, while Documentation and Testing issues were almost mapped onto Maldonado et al. categories. We noticed how our content-based SATD categorization does not have a one-to-one mapping to lifecycle-based categories.

TABLE III
DISTRIBUTION OF SENTIMENT LABELS.

Category	Negative	(%)	Non-negative	Mixed	Total
Macro-categories in our taxonomy					
Functional issues	66	(49%)	67	2	135
Poor implementation choices	125	(29%)	294	7	426
Wait	41	(46%)	45	3	89
Deployment issues	1	(50%)	1	0	2
SATD comments outdated	1	(33%)	2	0	3
Partially implemented	29	(13%)	197	2	228
Testing issues	12	(33%)	24	0	36
Documentation issues	18	(33%)	36	0	54
Misalignment	6	(29%)	15	0	21
Categories of Maldonado et al. [2]					
Defect	42	(40%)	60	4	106
Design	214	(34%)	407	9	630
Documentation	16	(41%)	23	0	39
Implementation	13	(7%)	163	1	177
Test	14	(33%)	28	0	42
TOTAL	299	(30%)	681	14	994

B. How is the sentiment polarity distributed across different kinds of SATD annotations?

Following the methodology described in Section II-C, we label 998 SATD comments ($= 1038 - 40$, where 40 comments have been excluded as false positives, i.e., SATD comments that are not real SATD). Four comments have been further excluded as the authors could not reach an agreement regarding their sentiment polarity. Hence, for this question, we looked at 994 SATD comments out of 1038 in the original dataset. We report the resulting distribution of sentiment labels in Table III.

We observe that 299 of the 994 comments (30%) convey negative sentiment polarity and only 14 items are labeled as mixed. Developers mostly complain about "Functional issues," with 49% of comments (66 out of 135) conveying negative sentiment, e.g., "TODO: include the rowids!!!!" in HIBERNATE or "something is very wrong here" in COLUMBA. Similarly, developers appear annoyed by required changes being on hold: 46% of comments (41 out of 89) belonging to the "Wait" category contains negative sentiment, such as "turn of focus stealing (workaround should be removed in the future!)" in COLUMBA. Similarly to self-directed anger studied by Gachechiladze et al. [10] we also found cases in which developers blame themselves, e.g., "this is retarded. excuse me while I drool and make stupid noises" in JEDIT.

Negative sentiment is also found in 33% of "Documentation issues" (e.g., "TODO: are we intentionally eating all events? - tfm 20060203 document!" in ARGOUML) and "Testing issues" (e.g., "TODO enable some proper tests!!" in JMETER).

TABLE IV
STATISTICAL COMPARISON OF NEGATIVE POLARITY (OR>1 MEANS THAT THE PROPORTION IS SIGNIFICANTLY GREATER FOR THE LEFT-SIDE CATEGORY). NON-SIGNIFICANT PAIRS ARE OMITTED IN THE TABLE.)

Category 1	Category 2	p-value	OR
Functional issues	Partially implemented	<0.01	6.52
Functional issues	Documentation issues	0.04	2.43
Functional issues	Poor implementation choices	<0.01	2.30
Poor implementation choices	Partially implemented	<0.01	2.85
Wait	Partially implemented	<0.01	5.82
Wait	Poor implementation choices	0.02	2.05
Testing issues	Partially implemented	0.02	3.41
Documentation issues	Partially implemented	0.03	2.67

As for "Poor implementation choices", which is the most frequently observed macro-category in our taxonomy with 426 comments, we observe 29% of negative sentiment comments (e.g., "TODO: terrible implementation!" in HIBERNATE).

When reporting a partial or non-implemented functionality developers are least likely to be negative (29 out of 228), e.g., "calculate the adjusted data area taking into account the 3D effect... this assumes that there is a 3D renderer, all this 3D effect is a bit of an ugly hack" in JFREECHART.

As for the remaining categories, they contain a very small number of comments so the results might bring anecdotal evidence and need to be further verified with a larger study.

The pairwise comparison of negative polarity in the macro-categories (see Table IV) confirms that negative sentiment mostly occurs in presence of bugs or the need to wait to see an issue resolved. Specifically, comments in "Functional issues" and "Wait" appear significantly more negative than comments labeled as "Partially implemented" (Odds Ratio equal to 6.25 and 5.82, respectively) and more than twice as negative than "Poor implementation choice." Moreover, statistical analysis confirms that comments reporting partial implementation are the least negative, compared to the other categories.

Table III also reports the sentiment polarity with respect to the original classification of Maldonado et al. [2]. We note that implementation debt is the only category with the percentage of comments having a sentiment polarity $< 30\%$. Moreover, in line with our results, developers are negative about both documentation and test debt (41% and 33% respectively). Surprisingly, in the defect category we found 42 over 106 (33%) comments expressing a negative sentiment, less than the one observed in our "Functional Issues" category (49%), probably due to the presence of comments highlighting that a piece of functionality is only partially implemented in the defect debt category by Maldonado et al. [2].

RQ_2 summary: SATD about functional issues conveys more negative sentiment. Also, being "on-hold" for various reasons that do not depend on themselves, make developers communicating negative sentiment. Developers are, instead, more neutral when reporting poor implementation choices, misalignment, or documentation/testing issues.

C. To what extent do SATD annotations belonging to different categories contain additional details?

TABLE V
DISTRIBUTION OF DIMENSIONS USED BY DEVELOPERS TO ANNOTATE TECHNICAL DEBT OVER MACRO-CATEGORIES.

Category	Component	Name	Bug id	URL	Date
Functional issues	47 (35%)	12 (9%)	11 (8%)	1 (1%)	9 (7%)
Poor implementation choices	152 (35%)	48 (11%)	5 (1%)	0	16 (4%)
Wait	21 (24%)	4 (5%)	9 (10%)	2 (2%)	1 (1%)
Deployment issues	0	0	0	0	0
SATD comments outdated	1 (33%)	0	1 (33%)	0	0
Partially implemented	50 (22%)	22 (10%)	0	0	1 (¡ 1%)
Testing issues	7 (19%)	3 (8%)	0	0	0
Documentation issues	19 (35%)	11 (20%)	0	0	1 (2%)
Misalignment	7 (33%)	4 (19%)	0	0	0
TOT. (UNIQUE)	304 (30%)	104 (10%)	26 (3%)	3 (0.3%)	28 (3%)

Following the methodology described in Section II-D, we leverage our dataset to perform a conceptual replication of the work on task annotations by Storey et al. [14]. This analysis results are presented in Table V. While Storey et al. surveyed developers, we analyze SATD comments: while frequently mentioned by the developers surveyed by Storey et al., additional details rarely appear in our dataset. Specifically, 64% of developers from Storey et al. study declared to add references to classes/methods/plug-ins/modules, in our study we found this happening in 304 SATD comments (30%) which, although not as high as 60%, is a conspicuous fraction of the total. As for the authors' names, instead, only 10% of the SATD comments in our sample contain them, even if around 50% of developers explicitly added their name in the annotations.

This may be confirmed considering that only 12 out of 135 SATD comments in the "Functional issues" category clearly report the name. However, about half of the SATD comments referring to a name fall into the "Poor implementation choices" category. One possibility is that during code reviewing processes, reviewers may identify the presence of wrong decisions and highlight them as source code comments.

Moving our attention to the inclusion of bug identifiers, 42% of the comments containing them belong to the "Functional issues" category, however, a non-negligible percentage (33%) concerns the "Wait" category. This is not surprising since developers may introduce a workaround due to a bug that needs to be fixed in the same project or in a third-party library being used. As regards the former, consider the SATD comment: "// TODO : YUCK!!! fix after HHH-1907 is complete" in HIBERNATE, while for the latter in ARGOUML we found a comment stating: "[...] NOTE: This is temporary and will go away [...] http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4714232" in which the bug is in java.awt library.

Finally, looking both at dates and URLs, percentages in the dataset are very low compared to those reported in a survey by Storey et al. [14] (3% and 0.3% vs 19%, and 30%). A possible interpretation is that unlikely as stated in the survey, developers assume redundant introducing signature and date (as such information is available in the versioning system anyway). Nevertheless, having them explicitly stated in the source code makes the accountability and tracing more evident.

RQ_3 summary: The addition of details such as bug identifiers and names is not so frequent when reporting TD in source code comments. However, developers tend to mention classes and methods more frequently, possibly to improve traceability and supporting themselves/others in addressing the SATD.

IV. Discussion

Sentiment in SATD: a proxy for priority? Negative sentiment is most frequently associated with reporting functional issues but also with developers waiting for issue resolution that does not depend on themselves. In other words, developers perceive both the presence of bugs and being "on-hold" as more annoying than other problems, such as partial implementations, testing, and documentation issues. While we acknowledge the need for further investigation of sentiment in SATD, e.g., on a larger dataset, we believe these findings already have actionable implications. Specifically, the amount of negativity observed in the "Functional issues" category suggests that developers should prioritize bug fixing over other issues, such as the implementation of missing functionality. This is also in line with previous findings by Mäntylä et al. [11] reporting more negativity for bugs and more positive sentiment for feature implementation requests. i.e., if a SATD comment reflects a functional issue, then we need to report it in an issue tracker or invoke automatic program repair.

As for waiting, the high negative sentiment associated with being "on-hold" might be interpreted as an indication of a blocking issue urgently requiring attention. This is in line with previous findings by Ortu et al. [32] reporting a positive correlation between negative sentiment and issue fixing time. Along the same line, Mäntylä et al. [11] reported higher emotional activation as the issue resolution time increases, as well as higher arousal in high priority bug reports, thus indicating a presence of emotions with high activation and negative polarity, such as stress. As such, the presence of negative sentiment can be used as a proxy for automatic identification and prioritization of critical, blocking issues, which might require the interventions of peers. Secondly, the information in the classification can be used to assist in the fine-grained problem of SATD prioritization.

Supporting developers in effective SATD comment writing: the role of sentiment. Based on the results of the sentiment analysis study, we believe that providing immediate feedback on the negative tone during comment-writing could support developers in more effective collaboration. Specifically, early detection of harsh or hostile sentiment could not only enable discovering code of conduct violations [33] but also support developers towards effective communication. A SATD sentiment analyzer could prompt developers warnings to highlight toxicity conveyed by their comments and possibly suggest re-tuning their writing, to avoid irritating their peers.

Our vision is also supported by previous findings. Motivated by developers reporting stress due to aggressive communicative behavior in open source communities, Raman et al. [34] investigated the possibility to automatically detect and mitigate such unhealthy interactions. Steinmacher et al. [35], instead, showed the impact of social barriers in attracting new contributors to open-source projects. Further studies investigated the impact of sentiment in collective knowledge-building: Calefato et al. [36] found a higher probability of fulfilling information-seeking goals on Stack Overflow when questions are formulated using a neutral style, while Choi et al. [37] found that positive, welcoming tone and constructive criticism is beneficial for online collaboration in Wikipedia.

The evaluation of the SE-specific publicly available sentiment analysis tools we performed (see Section II) indicated that a fine-tuning is needed before existing tools can be reliably used. Our gold standard for sentiment annotation in SATD

represents the first step towards this goal. Furthermore, being able to reliably identify and distinguish hostile comments from non-toxic negative sentiment, as in reporting concerns due to a bug, is a crucial aspect to take into account in performing such fine-tuning to avoid marking non-toxic comments for moderation. By releasing our gold standard and guidelines for annotation, we hope to stimulate further research on negativity detection in SATD.

References perceived as important in comments [14], but not widely used in SATD comments. Based on the results of our study we envision the emergence of tools supporting and guiding the authors towards adding proper references and information while adding SATD. While previous work by Storey et al. [14] stressed the perceived importance of various forms of references in task annotations, they occur much less frequently than one would expect. Depending on the category of our taxonomy, developers should have clear guidelines to properly document SATD, such as:

- for "Functional issues" SATD: open bug reports in the issue tracker and reference them in the comment;
- for "Wait", whenever possible, reference the origin of the on-hold, i.e., either an issue to be fixed or artifacts (other classes or methods) to be updated

Tool support could be developed to automatically detect introduction/change of SATD comments, and generate a date and signature for it, since half of developers in the study by Storey et al. include both their names and dates during task annotations. Similarly, automated support could be provided to reference/open an issue every time a Functional SATD is detected. Also, when "on-hold" SATD comment is automatically detected [30], developers may be guided to add a reference to a proper source. By helping to achieve properly structured SATD comments (depending on their type) with suitable references, not only those comments may become more traceable and understandable, but the available information will also help to better drive their manual (or semi-automated) resolution.

V. Threats to Validity

Threats to *construct validity* concern the relationship between theory and observation. One threat is how the comments are classified in RQ_1. Our knowledge of the analyzed systems may not be as deep as those of the original developers. To mitigate this threat, we looked not only at the comments but also at the corresponding source code when this was needed. A relevant threat for RQ_2 is related to how "sentiment" is perceived by annotators but may not match the actual sentiment of developers. For what possible, subjectiveness in RQ_1 and RQ_2 has been mitigated by establishing clear coding guidelines, and by doing initial joint sessions. Furthermore, we resolved all disagreements through a discussion during plenary meetings involving all the annotators. For sentiment labeling, we also measured the extent to which we could have reached an agreement by chance using inter-rater agreement metrics.

Threats to *internal validity* are related to factors internal to our study that can affect our results. Although we created a relatively large and statistically significant sample, we cannot exclude that our sampling strategy is weakly representative of the studied dataset. In particular, we sampled our dataset starting from the data and categories of Maldonado et al., so we might have inherited representativeness threats from the original study. Finally, measurement imprecision in RQ_3 has been mitigated, where it matters, through manual analysis.

Threats to *external validity* concern the generalizability of our findings. The qualitative nature of the study (especially RQ_1) and the need for manual inspection for all three research questions do not make a large-scale analysis feasible. Therefore, although the sample is statistically significant, it may not generalize to further projects and programming languages different from Java. For RQ_1, although we reached saturation when identifying categories, we cannot exclude that new categories would emerge when looking at further datasets.

VI. Related Work

In the following, we discuss relevant literature related to (i) studies about TD and SATD, and (ii) sentiment analysis in software development.

A. Technical Debt and Self-Admitted Technical Debt

In the past years, the research community empirically studied TD and SATD. Seaman and Guo [38], Kruchten et al. [39], Brown et al. [40], and Alves et al. [41] made different considerations about "technical debt" highlighting that TDs are a communication media among developers and managers to discuss and address development issues. Furthermore, Lim et al. [42] highlighted that TD introduction is mostly intentional, and Ernst et al. [43] pointed out how TD awareness is a cornerstone for TD management. Zazworka et al. [44], instead, highlighted the need for proper handling and identification of TD to reduce their negative impact on software quality.

By looking at source code comments in open source projects Potdar and Shihab [1] found that developers tend to "self-admit" TD. In a follow-up study, Maldonado and Shihab [7] developed an approach that by using 62 patterns identifies whether or not a comment is an SATD along with such categories as defect, design, documentation, requirement, and test debts. Bavota and Russo [8], instead, have refined the above classification providing a taxonomy featuring 6 higher-level TD categories properly specialized into 11 sub-categories. Our work differs from that by Potdar and Shihab [1] and Bavota and Russo [8] in that we focus on the content reported in the SATD without considering the development life-cycle in which the SATD may be mapped.

Concerning the SATD classification, Maipradit et al. [30], introduced the concept of "on-hold" SATD i.e., comments expressing a condition indicating that a developer is waiting for an event internal or external to the project under development. As a follow-up study, Maipradit et al. [45], built a classifier aimed at detecting on-hold SATD with an average AUC of 0.97. Finally, Fucci et al. [46], conjectured that "self-admission" may not necessarily mean that the comment has been introduced by whoever has written or changed the source code. Their results highlight that SATD comments are mainly

introduced by developers having a high level of ownership on the SATD-affected source code.

As regards the impact of SATD and its management, Wehaibi et al. [47] found that SATD leads to complex changes in the future, while Kamei et al. [48], highlighted that $\simeq 42\%$ of TD incurs positive interest. From a different perspective, Zampetti et al. [49] developed an approach for recommending when a design TD has to be admitted.

Differently from previous work, we focused our attention on the SATD content, i.e., what developers usually annotate about TD, as well as how they communicate the presence of this temporary solution, i.e., sentiment and external references.

The research community has also focused on SATD removal. Maldonado et al. [4] found that there is a high percentage of SATD being removed even if their survivability varies by project. Zampetti et al. [5], instead, studied the relationship between comment removals and changes applied to the affected source code. They found how SATD can be either removed through focused changes (e.g., to conditional statements), but also by rewriting/replacing substantial portions of source code. In a follow-up study, Zampetti et al. [6] proposed SARDELE, a multi-level classifier able to recommend six SATD removal strategies using a deep learning approach. We believe that a more focused analysis of the SATD content like the one done in our work could help to refine such approaches, allowing for more actionable suggestions.

B. Sentiment Analysis in Software Development

Recently, a trend has emerged and consolidated to leverage sentiment analysis in empirical software engineering research [50]. Murgia et al. [25] presented an early exploratory study of emotions in software artifacts. By manually labeling issues from the Apache Software Foundation, they found that developers feel and report a variety of emotions, including gratitude, joy, and sadness. Ortu et al. [32], instead, investigated the correlation between sentiment in issues and their fixing time showing how issues with negative polarity, e.g., sadness, have a longer fixing time. On the same line, Mäntylä et al. [11] performed a correlation study between emotions and bug priority to derive symptoms of productivity loss and burnout. By looking at issue tracking comments they mined emotions and used them to compute Valence (i.e., sentiment polarity), Arousal (i.e., sentiment intensity), and Dominance (the sensation of being in control of a situation). Their findings highlight that bug reports are associated with a more negative Valence, and issue priority positively correlates with the emotional activation, with higher priority correlating with higher arousal. Differently from the previous correlation studies, our study investigates how developers communicate the presence of technical debt by manually labeling the sentiment inside SATD comments. Our results confirm previous findings in terms of using sentiment as a proxy for problems and priority in the software development process.

Furthermore, researchers in requirements engineering use sentiment analysis as a source of information for requirements classification towards supporting software maintenance and evolution. Panichella et al. [51] applied sentiment analysis for classifying user reviews in Google Play and Apple Store, while Maalej et al. [52] leveraged several text-based features, including sentiment, for automatically classifying app reviews into four categories, namely bug reports, feature requests, user experiences, and text ratings.

As far as negative emotions are concerned, Gachechiladze et al. [10] looked at the anger and its direction in collaborative software development, envisioning the tools detecting the anger target in developers' communication, by distinguishing between anger towards *self*, *others*, and *object*. As a preliminary step towards this goal, they created a manually annotated dataset of 723 sentences from the Apache issue reports and used it to train a supervised classifier for anger detection. Similarly to this study, we focus on negative emotion confirming that their detection and modeling can serve as a proxy for problems occurring in the software development process.

A complementary line of research considers biometric measurements to assess software developers' emotional states rather than texts authored by them [53], [54].

VII. Conclusion

In this paper, we have studied Self-Admitted Technical Debt (SATD) comments: what kind of information is being exchanged and how is this being done. We have manually analyzed 1038 SATD comments and constructed a taxonomy of 41 categories with 9 top-level categories: functional issues, poor implementation choices, waiting, deployment issues, outdated SATD comments, partially/not implemented functionality, testing issues, documentation issues, and misalignment-related problems. Not surprisingly, most SATD comments pertain to poor implementation choices and partially/not implemented functionality. Compared to a previous classification of Maldonado et al. [2] we observe that while some categories, e.g., Testing, can be mapped to the corresponding categories of Maldonado et al. [2], other categories, e.g., functional issues, poor implementation choices, and partially/not implemented functionality, are spread over multiple categories in Maldonado et al., i.e., defects, design, and implementation.

We then analyzed to what extent is negative sentiment expressed in SATD comments of different categories. We observed that SATD about functional issues conveys more negative sentiment, but also being "on-hold" for various reasons that do not depend on themselves make developers expressing themselves in a negative way. Developers are, instead, more neutral for poor implementation choices, misalignment, or partially/not implemented functionality. In the title parlance "waiting around" is perceived as more problematic or more important than "job half-done" (partial implementation). This calls for further research targeting bugs and on-hold TD expressed in SATD comments.

Finally, we found how, despite previous work showed how developers perceive as important adding external references to comments, we found them only in a minority of the analyzed cases. This calls for recommenders aimed at helping developers to better annotating source code in presence of TD.

REFERENCES

[1] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*, 2014, pp. 91–100.

[2] E. da S. Maldonado, E. Shihab, and N. Tsantalis, "Using natural language processing to automatically detect self-admitted technical debt," *IEEE Trans. Software Eng.*, vol. 43, no. 11, pp. 1044–1062, 2017.

[3] X. Ren, Z. Xing, X. Xia, D. Lo, X. Wang, and J. Grundy, "Neural network-based detection of self-admitted technical debt: From performance to explainability," *ACM Trans. Softw. Eng. Methodol.*, vol. 28, no. 3, p. 15, 2019.

[4] E. da S. Maldonado, R. Abdalkareem, E. Shihab, and A. Serebrenik, "An empirical study on the removal of self-admitted technical debt," in *ICSME*, 2017, pp. 238–248.

[5] F. Zampetti, A. Serebrenik, and M. Di Penta, "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, Gothenburg, Sweden, May 28-29, 2018*, 2018, pp. 526–536.

[6] F. Zampetti, A. Serebrenik, and M. Di Penta, "Automatically learning patterns for self-admitted technical debt removal," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2020, pp. 355–366.

[7] E. da S. Maldonado and E. Shihab, "Detecting and quantifying different types of self-admitted technical debt," in *7th IEEE International Workshop on Managing Technical Debt, MTD@ICSME 2015, Bremen, Germany, October 2, 2015*, 2015, pp. 9–15.

[8] G. Bavota and B. Russo, "A large-scale empirical study on self-admitted technical debt," in *MSR*, 2016, pp. 315–326.

[9] D. Pletea, B. Vasilescu, and A. Serebrenik, "Security and emotion: Sentiment analysis of security discussions on github," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 348351.

[10] D. Gachechiladze, F. Lanubile, N. Novielli, and A. Serebrenik, "Anger and its direction in collaborative software development," in *Proceedings of the 39th International Conference on Software Engineering: New Ideas and Emerging Results Track*, ser. ICSE-NIER '17. IEEE Press, 2017, p. 1114.

[11] M. Mäntylä, B. Adams, G. Destefanis, D. Graziotin, and M. Ortu, "Mining valence, arousal, and dominance: Possibilities for detecting burnout and productivity?" in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 247258.

[12] G. Uddin and F. Khomh, "Opiner: An opinion search and summarization engine for apis," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, p. 978983.

[13] B. Lin, F. Zampetti, G. Bavota, M. Di Penta, and M. Lanza, "Pattern-based mining of opinions in q a websites," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 548–559.

[14] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, and J. Singer, "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*, ser. ICSE '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 251260. [Online]. Available: https://doi.org/10.1145/1368088.1368123

[15] F. Shull, J. C. Carver, S. Vegas, and N. Juristo Juzgado, "The role of replications in empirical software engineering," *Empir. Softw. Eng.*, vol. 13, no. 2, pp. 211–218, 2008. [Online]. Available: https://doi.org/10.1007/s10664-008-9060-1

[16] D. Spencer, *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[17] K. R. Scherer, T. Wranik, J. Sangsue, V. Tran, and U. Scherer, "Emotions in everyday life: probability of occurrence, risk factors, appraisal and reaction patterns," *Social Science Information*, vol. 43, no. 4, pp. 499–570, 2004. [Online]. Available: https://doi.org/10.1177/0539018404047701

[18] N. Novielli, D. Girardi, and F. Lanubile, "A benchmark study on sentiment analysis for software engineering research," in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 364375. [Online]. Available: https://doi.org/10.1145/3196398.3196403

[19] M. R. Islam and M. F. Zibran, "Sentistrength-se: Exploiting domain specificity for improved sentiment analysis in software engineering text," *Journal of Systems and Software*, vol. 145, pp. 125 – 146, 2018. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121218301675

[20] F. Calefato, F. Lanubile, F. Maiorano, and N. Novielli, "Sentiment Polarity Detection for Software Development," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1352–1382, 2018.

[21] T. Ahmed, A. Bosu, A. Iqbal, and S. Rahimi, "SentiCR: A customized sentiment analysis tool for code review interactions," *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 106–111, 2017.

[22] N. Novielli, F. Calefato, F. Lanubile, and A. Serebrenik, "Assessment of off-the-shelf SE-specific sentiment analysis tools: An extended replication study," *Empir. Softw. Eng.*, vol. 26, 2021.

[23] N. Novielli, F. Calefato, D. Dongiovanni, D. Girardi, and F. Lanubile, "Can We Use SE-specific Sentiment Analysis Tools in a Cross-Platform Setting?" *Proceedings - 2020 IEEE/ACM 17th International Conference on Mining Software Repositories, MSR 2020*, pp. 158–168, 2020.

[24] K. Krippendorff, *Content analysis: An introduction to its methodology*. Sage, 2012.

[25] A. Murgia, P. Tourani, B. Adams, and M. Ortu, "Do developers feel emotions? an exploratory analysis of emotions in software artifacts," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 262271. [Online]. Available: https://doi.org/10.1145/2597073.2597086

[26] R. G. Newcombe, "Interval estimation for the difference between independent proportions: comparison of eleven methods," *Statistics in Medicine*, vol. 17, no. 8, pp. 873–890, 1998. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/%28SICI%291097-0258%2819980430%2917%3A8%3C873%3A%3AAID-SIM779%3E3.0.CO%3B2-I

[27] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 57, no. 1, pp. 289–300, 1995.

[28] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *Software Maintenance, 2003. ICSM 2003. Proceedings. International Conference on*. IEEE, 2003.

[29] F. Ebert, F. Castor, N. Novielli, and A. Serebrenik, "Communicative intention in code review questions," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 519–523.

[30] R. Maipradit, C. Treude, H. Hata, and K. Matsumoto, "Wait for it: identifying on-hold self-admitted technical debt," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3770–3798, 2020.

[31] B. Fluri, M. Wursch, and H. C. Gall, "Do code and comments co-evolve? on the relation between source code and comment changes," in *14th Working Conference on Reverse Engineering (WCRE 2007)*. IEEE, 2007, pp. 70–79.

[32] M. Ortu, B. Adams, G. Destefanis, P. Tourani, M. Marchesi, and R. Tonelli, "Are bullies more productive? empirical study of affectiveness vs. issue fixing time," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, 2015, pp. 303–313.

[33] P. Tourani, B. Adams, and A. Serebrenik, "Code of conduct in open source projects," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 24–33.

[34] N. Raman, M. Cao, Y. Tsvetkov, C. Kästner, and B. Vasilescu, "Stress and burnout in open source: Toward finding, understanding, and mitigating unhealthy interactions," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ser. ICSE-NIER '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 5760. [Online]. Available: https://doi.org/10.1145/3377816.3381732

[35] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles, "Social barriers faced by newcomers placing their first contribution in open source software projects," in *CSCW 2015*, ser. CSCW '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 13791392. [Online]. Available: https://doi.org/10.1145/2675133.2675215

[36] F. Calefato, F. Lanubile, and N. Novielli, "How to ask for technical help? evidence-based guidelines for writing questions on stack overflow," *Inf. Softw. Technol.*, vol. 94, no. C, p. 186207, Feb. 2018.

[37] B. Choi, K. Alexander, R. E. Kraut, and J. M. Levine, "Socialization tactics in wikipedia and their effects," in *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, ser. CSCW '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 107116. [Online]. Available: https://doi.org/10.1145/1718918.1718940

[38] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," *Advances in Computers*, 2011.

[39] P. Kruchten, R. L. Nord, I. Ozkaya, and D. Falessi, "Technical debt: towards a crisper definition report on the 4th international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, 2013.

[40] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. L. Nord, I. Ozkaya, R. S. Sangwan, C. B. Seaman, K. J. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the Workshop on Future of Software Engineering Research, FoSER 2010, at the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010*, G. Roman and K. J. Sullivan, Eds. ACM, 2010, pp. 47–52.

[41] N. S. R. Alves, L. F. Ribeiro, V. Caires, T. S. Mendes, and R. O. Spínola, "Towards an ontology of terms on technical debt," in *MTD*, 2014, pp. 1–7.

[42] E. Lim, N. Taksande, and C. Seaman, "A balancing act: what software practitioners have to say about technical debt," *IEEE software*, 2012.

[43] N. A. Ernst, S. Bellomo, I. Ozkaya, R. L. Nord, and I. Gorton, "Measure it? manage it? ignore it? software practitioners and technical debt," in *Foundations of Software Engineering*. ACM, 2015, pp. 50–60.

[44] N. Zazworka, M. A. Shaw, F. Shull, and C. B. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the 2nd Workshop on Managing Technical Debt, MTD 2011, Waikiki, Honolulu, HI, USA, May 23, 2011*, 2011, pp. 17–23.

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

[46] G. Fucci, F. Zampetti, A. Serebrenik, and M. Di Penta, "Who (self) admits technical debt?" in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 672–676.

[47] S. Wehaibi, E. Shihab, and L. Guerrouj, "Examining the impact of self-admitted technical debt on software quality," in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 179–188.

[48] Y. Kamei, E. d. S. Maldonado, E. Shihab, and N. Ubayashi, "Using analytics to quantify interest of self-admitted technical debt." in *QuASoQ/TDA@ APSEC*, 2016, pp. 68–71.

[49] F. Zampetti, C. Noiseux, G. Antoniol, F. Khomh, and M. Di Penta, "Recommending when design technical debt should be self-admitted," in *ICSME*, 2017.

[50] N. Novielli and A. Serebrenik, "Sentiment and emotion in software engineering," *IEEE Softw.*, vol. 36, no. 5, pp. 6–9, 2019. [Online]. Available: https://doi.org/10.1109/MS.2019.2924013

[51] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "How can i improve my app? classifying user reviews for software maintenance and evolution," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 281–290.

[52] W. Maalej, Z. Kurtanovic, H. Nabil, and C. Stanik, "On the automatic classification of app reviews," *Requirements Engineering*, vol. 21, pp. 311–331, 2016.

[53] S. C. Müller and T. Fritz, "Stuck and frustrated or in flow and happy: Sensing developers' emotions and progress," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, A. Bertolino, G. Canfora, and S. G. Elbaum, Eds. IEEE Computer Society, 2015, pp. 688–699. [Online]. Available: https://doi.org/10.1109/ICSE.2015.334

[54] D. Girardi, N. Novielli, D. Fucci, and F. Lanubile, "Recognizing developers' emotions while programming," in *ICSE*, 2020, p. 666677.