Noname manuscript No.
(will be inserted by the editor)

Wait For It: Identifying "On-Hold" Self-Admitted Technical Debt

Rungroj Maipradit · Christoph Treude · Hideaki Hata · Kenichi Matsumoto

Received: date / Accepted: date

Abstract Self-admitted technical debt refers to situations where a software developer knows that their current implementation is not optimal and indicates this using a source code comment. In this work, we hypothesize that it is possible to develop automated techniques to understand a subset of these comments in more detail, and to propose tool support that can help developers manage self-admitted technical debt more effectively. Based on a qualitative study of 333 comments indicating self-admitted technical debt, we first identify one particular class of debt amenable to automated management: on-hold self-admitted technical debt (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. We then design and evaluate an automated classifier which can identify these on-hold instances with an area under the receiver operating characteristic curve (AUC) of 0.98 as well as detect the specific conditions that developers are waiting for. Our work presents a first step towards automated tool support that is able to indicate when certain instances of self-admitted technical debt are ready to be addressed.

Keywords Self-admitted technical debt · qualitative study · classification

1 Introduction

The metaphor of technical debt is used to describe the trade-off many software developers face when developing software: how to balance near-term value with

Rungroj Maipradit · Hideaki Hata · Kenichi Matsumoto
Nara Institute of Science and Technology
E-mail: maipradit.rungroj.mm6@is.naist.jp, hata@is.naist.jp, matumoto@is.naist.jp

Christoph Treude
University of Adelaide
E-mail: christoph.treude@adelaide.edu.au

long-term quality (Ernst et al., 2015). Practitioners use the term technical debt as a synonym for "shortcut for expediency" (McConnell, 2007) as well as to refer to bad code and inadequate refactoring (Kniberg, 2013). Technical debt is widespread in the software domain and can cause increased software maintenance costs as well as decreased software quality (Lim et al., 2012).

In many cases, developers know when they are about to cause technical debt, and they leave documentation to indicate its presence (Maldonado et al., 2017b). This documentation often comes in the form of source code comments, such as "`TODO: This method is too complex, [let's] break it up`" and "`TODO no methods yet for getClassname`".[1] Previous work (Ichinose et al., 2016) has explored the use of visualization to support the discovery and removal of self-admitted technical debt, incorporating gamification mechanisms to motivate developers to contribute to the debt removal. Current research is largely focused on the detection and classification of self-admitted technical debt, but has spent less effort on approaches to address the debt automatically, likely because work on the detection and classification is still very recent.

Previous work (Maldonado et al., 2017b) has developed an approach based on natural language processing to automatically detect self-admitted technical debt comments and to classify them into either design or requirement debt. Self-admitted design debt encompasses comments that indicate problems with the design of the code while self-admitted requirement debt includes all comments that convey the opinion of a developer suggesting that the implementation of a requirement is incomplete. In general terms, design debt can be resolved by refactoring whereas requirement debt indicates the need for new code.

In this work, we hypothesize that it is possible to use automated techniques based on natural language processing to understand a subset of the technical debt categories identified in previous work in more detail, and to propose tool support that can help developers manage self-admitted technical debt more effectively. We make three contributions:

– A qualitative study on the removal of self-admitted technical debt. To understand what kinds of technical debt could be addressed or managed automatically, we annotated a statistically representative sample of instances of self-admitted technical debt removal from the data set made available by the authors of previous work (Maldonado et al., 2017a). While the focus of our annotators was on the identification of instances of self-admitted technical debt that could be automatically addressed, as part of this annotation, we also performed a partial conceptual replication (Shull et al., 2008) of recent work by Zampetti et al. (2018),[2] who found that a large percentage of self-admitted technical debt removals occur accidentally. We were able to confirm this finding: in 58% of the cases in our sample, the self-

[1] Examples from ArgoUML and Apache Ant, respectively (Maldonado et al., 2017b).

[2] Note that Zampetti et al. (2018) was published after we commenced this project, i.e., we do not use their data.

```
// TODO the following code is copied from AbstractSimpleBeanDefinitionParser
// it can be removed if ever the doParse() method is not final!
// or the Spring bug http://jira.springframework.org/browse/SPR-4599 is resolved
```

Fig. 1: Motivating Example[3]

admitted technical debt was not actually addressed, but the admission was simply removed. This finding is also in line with findings from Bazrafshan and Koschke (2013) who reported a large number of accidental removals of cloned code. Zampetti et al. (2018) further reported that in removing self-admitted technical debt comments, developers tend to apply complex changes. Our work indirectly confirms this by finding that a majority of changes which address self-admitted technical debt could not easily be applied to similar debt in a different project.

– The definition of *on-hold self-admitted technical debt* (on-hold SATD). Our annotation revealed one particular class of self-admitted technical debt amenable to automated management: on-hold SATD. We define on-hold SATD as self-admitted technical debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere. Figure 1 shows an example of on-hold SATD from the Apache Camel project. The developer is waiting for an external event (the visibility of `doParse()` changing or an external bug being resolved) and the comment admitting the debt is therefore on hold.
– The design and evaluation of a classifier for self-admitted technical debt. Since software developers must keep track of many events and updates in any software ecosystem, it is unrealistic to assume that developers will be able to keep track of all self-admitted technical debt and of events that signal that certain self-admitted technical debt is now ready to be addressed. To support developers in managing self-admitted technical debt, we designed a classifier which can automatically identify those instances of self-admitted technical debt which are on hold, and detect the specific events that developers are waiting for. Our classifier achieves an area under the receiver operating characteristic curve (AUC) of 0.98 for the identification, and 90% of the specific conditions are detected correctly. This is a first step towards automated tool support that can recommend to developers when certain instances of self-admitted technical debt are ready to be addressed.

The remainder of this paper is structured as follows: In Section 2, we present our research questions and the methods that we used for collecting and analyzing data for the qualitative study. The findings from this qualitative study are presented in Section 3. Section 4 describes the design of our classifier

[3] cf. `https://github.com/apache/camel/blob/53177d55053a42f6fd33434895c60615713f4b78/components/camel-spring/src/main/java/org/apache/camel/spring/handler/BeanDefinitionParser.java`

to identify on-hold SATD, and we present the results of our evaluation of the classifier in Section 5. Section 6 discusses the implications of this work, before Section 7 highlights the threats to validity and Section 8 summarizes related work. Section 9 outlines the conclusions and highlights opportunities for future work.

2 Research Methodology

In this section, we detail our research questions as well as the methods for data collection and analysis used in our qualitative study. We also describe the data provided in our online appendix.

2.1 Research Questions

Our research questions focus on identifying how self-admitted technical debt is typically removed and whether the fixes applied to this debt could be applied to address similar debt in other projects. To guide our work, we first ask about the different kinds of self-admitted technical debt that can be found in our data (RQ1.1), whether the commits which remove the corresponding comments actually fix the debt (RQ1.2), and if so, what kind of fix has been applied (RQ1.3). To understand the removal in more detail, we also investigate whether the removal was the primary reason for the commit (RQ1.4), before investigating the subset of self-admitted technical debt that could be managed automatically (RQ1.5). Based on the definition of on-hold SATD which emerged from our qualitative study to answer these questions, we then investigate its prevalence (RQ1.6) and the accuracy of automated classifiers to identify this particular class of self-admitted technical debt (RQ2.1) and its specific sub-conditions (RQ2.2):

RQ1 How do developers remove self-admitted technical debt?
- RQ1.1 What kinds of self-admitted technical debt do developers indicate?
- RQ1.2 Do commits which remove the comments indicating self-admitted technical debt actually fix the debt?
- RQ1.3 What kinds of fixes are applied to address self-admitted technical debt?
- RQ1.4 Is the removal of self-admitted technical debt the primary reason for the commits which remove the corresponding comments?
- RQ1.5 Could the fixes applied to address self-admitted technical debt be applied to address similar debt in other projects?
- RQ1.6 How many of the comments indicating self-admitted technical debt contain a condition to specify that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere?

RQ2 How accurately can our classifier automatically identify on-hold SATD?
- RQ2.1 What is the best performance of our classifier to automatically identify on-hold SATD?

Table 1: Data set

project	**SATD removal commits**	**sample**
Apache Camel	987	128
Apache Tomcat	910	125
Apache Hadoop	370	52
Gerrit Code Review	133	19
Apache Log4j	107	9
Total	2,507	333

RQ2.2 How well can our classifier automatically identify the specific conditions in on-hold SATD?

2.2 Data Collection

To obtain data on the removal of self-admitted technical debt, we used the online appendix of Maldonado et al. (2017a) as a starting point. In their work, Maldonado et al. conducted an empirical study on five open source projects to examine how self-admitted technical debt is removed, who removes it, for how long it lives in a project, and what activities lead to its removal. They make their data available in an online appendix[4], which contains 2,599 instances of a commit removing self-admitted technical debt. After removing duplicates, 2,507 instances remain. The first two columns of Table 1 show the number of commits for each of the five projects available in this data set. Note that as a consequence of reusing this data set, we are implicitly also reusing Maldonado et al. (2017a)'s definition of technical debt as well as their interpretation of what constitutes debt removal.

Based on this data set of commits which removed a comment indicating self-admitted technical debt (after removing duplicates), we created a statistically representative and random sample (confidence level 95%, confidence interval 5 of 333 commits. The last column of Table 1 shows the number of commits from each project in our sample.

2.3 Data Analysis

To answer our first research question "How do developers remove self-admitted technical debt?" and its sub-questions, we performed a qualitative study on the sample of 333 commits which had removed self-admitted technical debt according to the data provided by Maldonado et al. (2017a).

In the first step, the second and third author of this paper independently analyzed twenty commits from the sample to determine appropriate questions to be asked during the qualitative study, aiming to obtain insights into how

[4] `http://das.encs.concordia.ca/uploads/2017/07/maldonado_icsme2017.zip`

Table 2: Qualitative annotation schema

question	**answers**	**motivation**
Does the comment represent self-admitted technical debt?	yes/no	Observation that some comments that Maldonado et al. (2017a) had automatically identified as self-admitted technical debt did not actually constitute debt
RQ1.1 What kind of self-admitted technical debt was it?	open	Distinguishing different kinds of self-admitted technical debt, with the ultimate goal of identifying ones that can be addressed automatically
RQ1.2 Did the commit fix the self-admitted technical debt?	yes/no	Observation that commits which remove self-admitted technical debt do not necessarily fix the debt, as also found by Zampetti et al. (2018)
RQ1.3 What kind of fix was it?	open	Distinguishing different kinds of fixes for self-admitted technical debt, to study whether fixes could be applied automatically
RQ1.4 Was removing the self-admitted technical debt the primary reason for the commit?	yes/no	Observation that even for those commits which addressed self-admitted technical debt, this was not necessarily their main purpose
RQ1.5 Could the same fix be applied to similar self-admitted technical debt in a different project?	possibly/no	Identifying fixes that could potentially be applied automatically
RQ1.6 Does the self-admitted technical debt comment include a condition?	yes/no	Exploring the phenomenon of on-hold SATD—which emerged from answering the previous question—in more detail

developers remove self-admitted technical debt and to identify the kinds of debt that could be addressed or managed automatically. After several iterations and meetings, the second and third author agreed on seven questions that should be answered for each of the 333 commits during the qualitative study. These questions along with their motivation and answer ranges are shown in Table 2.

The first author annotated all 333 commits following this annotation schema, and the second and third author annotated 50% of the data each, ensuring that each commit was annotated according to all seven questions by two researchers. Note that not all questions applied to all commits. For example, all instances which we classified as not representing self-admitted technical debt were not considered for future questions, and all commits which we classified as not fixing self-admitted technical debt were not considered for questions such as “Could the same fix be applied to similar Self-Admitted Technical Debt in a different project?”.

After the annotation, the first three authors conducted multiple meetings in which they determined consistent coding schemes for the two questions which allowed for open answers and collaboratively resolved all disagreements in the annotation until reaching consensus on all ratings. We report the initial agreement for each question before the resolution of disagreements as part of our findings in the next section.[5]

2.4 Online Appendix

Our online appendix contains descriptive information on the 333 commits which were labeled as removing self-admitted technical debt according to Maldonado et al. (2017a) along with our qualitative annotations in response to the seven questions. Our online appendix also includes the data set we use in testing and training our classifier. The appendix is available at `https://tinyurl.com/onholddebt`.

3 Qualitative Findings

In this section, we describe the findings derived from our qualitative study, separately for each sub-question of RQ1.

3.1 Initial Analysis

As shown in Figure 2, we found that not all commits which were automatically classified as removing self-admitted technical debt by the work of Maldonado et al. (2017a) actually removed a comment indicating debt. In some cases (9%)—indicated as N/A in Figure 2—the comment was not removed but only

[5] We calculated kappa values using `https://www.graphpad.com/quickcalcs/kappa1/`.

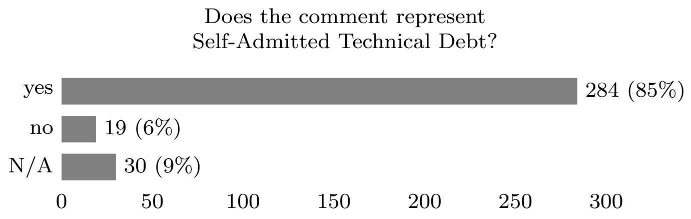

Fig. 2: Distribution of answers to "Does the comment represent Self-Admitted Technical Debt?". Initial agreement among the annotators before resolving disagreements: weighted kappa $\kappa = 0.820$ across 333 comments, i.e., "almost perfect" agreement (Viera and Garrett, 2005).

edited, and in other cases (6%), the comment had been incorrectly tagged as self-admitted technical debt, e.g., in the case of "`It is always a good idea to call this method when exiting an application`".

3.2 RQ1.1 What kinds of self-admitted technical debt do developers indicate?

Our first research question explores the different kinds of self-admitted technical debt found in our sample. Figure 3 shows the final result of our coding after consolidating the coding schema. The two most common kinds of debt in our sample are "functionality needed" (44%) and "refactoring needed" (17%). An example for the former is the comment "`TODO handle known multi-value headers`" while "`XXX move message resources in this package`" is an example for the latter. We also identified a number of clarification requests (15%), such as "`TODO: why not use millis instead of nano?`". We coded self-admitted technical debt comments that explicitly stated that they were temporary as workaround (8%), e.g., "`TODO this should subtract resource just assigned TEMPROARY`". We identified some comments which indicated that the developer was waiting for something (5%), such as "`TODO remove these methods if/when they are available in the base class!!!`". We will focus our discussion on these comments in the later parts of this paper. Finally, some comments which indicated technical debt describe bugs (4%, e.g., "`TODO this causes errors on shutdown...`") or focus on explaining the code (2%, e.g., "`some OS such as Windows can have problem doing rename IO operations so we may need to retry a couple of times to let it work`"). Note that for this annotation, we assigned exactly one code to each comment.

Previous classifications of self-admitted technical debt focused less on the actions required to remove the debt and more on what part of the software development lifecycle a debt item can be assigned to. For example, the categorisation of Maldonado and Shihab (2015) revealed five categories (design, defect, documentation, requirement, and test), and the categorisation of Bavota and Russo (2016) revealed the same five categories plus a sixth category called

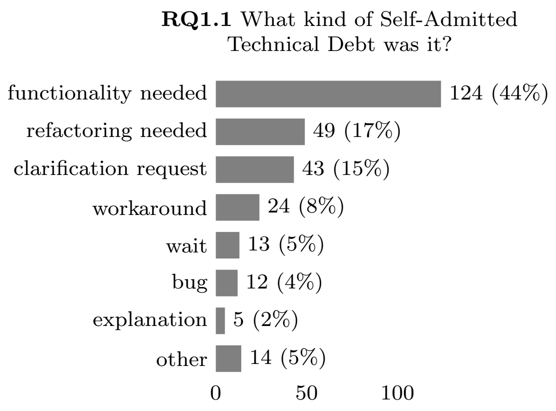

Fig. 3: Distribution of answers to “What kind of Self-Admitted Technical Debt was it?”. Initial agreement among the annotators before consolidating the coding schema: 45.07% across 284 comments.

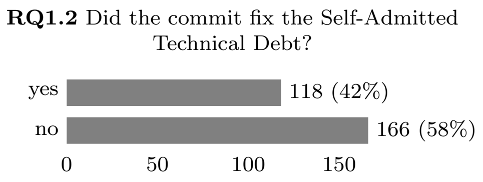

Fig. 4: Distribution of answers to “Did the commit fix the Self-Admitted Technical Debt?”. Initial agreement among the annotators before resolving disagreements: kappa $\kappa = 0.731$ across 284 comments, i.e., “substantial” agreement (Viera and Garrett, 2005).

“code”. In comparison, guided by our ultimate goal of identifying certain kinds of self-admitted technical debt which can be fixed automatically, our categorisation focuses more on what needs to be done in order to fix the debt, leading to categories such as “functionality needed” or “refactoring needed”.

3.3 RQ1.2 Do commits which remove the comments indicating self-admitted technical debt actually fix the debt?

For the majority of commits (58%) which removed the comment indicating technical debt, the commit did not actually fix the problem described in the comment, see Figure 4. Instead, these commits often removed the comment along with the surrounding code. These findings are in line with recent work by Zampetti et al. (2018) who found that between 20% and 50% of self-admitted technical debt is accidentally removed while entire classes or methods are dropped.

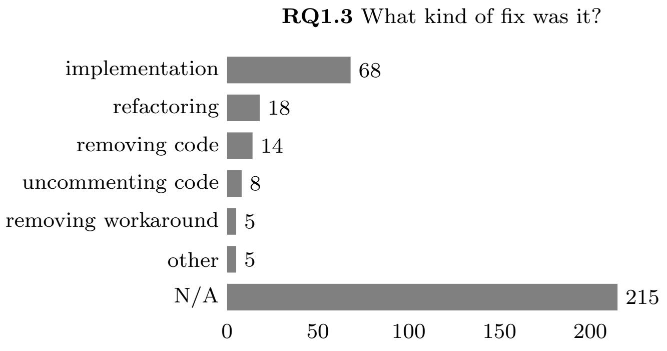

Fig. 5: Distribution of answers to “What kind of fix was it?”. Initial agreement among the annotators before consolidating the coding schema: 83.90% across 118 comments.

3.4 RQ1.3 What kinds of fixes are applied to address self-admitted technical debt?

In the cases where the commit fixed the self-admitted technical debt, we also coded the kind of fix that was applied. Figure 5 show the results of this coding: Debt was either fixed by implementing new code (58%), by refactoring existing code (15%), by removing code (12%), by uncommenting code that had been previously commented out (7%), or by removing a workaround (4%). Note that we used the commit message and/or related issue discussions to determine whether a change was meant to remove a workaround or was truly a refactoring. Other cases, such as uncommenting code, were easy to decide.

In the 215 cases where the commit does not fix the self-admitted technical debt, 30 commits do not remove the self-admitted technical debt comments or are tagged incorrectly, 19 comments do not represent self-admitted technical debt, and 166 commits do not fix self-admitted technical debt.

Our categorisation of the different kinds of fixes is at a slightly more coarse-granular level compared to that presented by Zampetti et al. (2018) who identified five categories (add/remove method calls, add/remove conditionals, add/remove try-catch, modify method signature, and modify return) in addition to “other”. In their categorisation, “other” accounts for 44% (339/779) of all instances. In comparison, our categorisation is less fine-grained, but contains fewer “other” cases.

Table 3 shows the relationship between the two coding schemes that emerged from our qualitative data analysis: one for the kinds of technical debt indicated in developer comments, and one for the kinds of fixes applied to this debt. Unsurprisingly, many instances where new functionality was needed were addressed by the implementation of said functionality, and cases where refactoring was needed were addressed by refactoring. Interestingly, all comments of developers explaining technical debt were removed without

Table 3: Types of Self-Admitted Technical Debt and the Corresponding Fixes. Each row represents a type of self-admitted technical debt, and each column represents a type of fix. The sum of each row and column indicates the overall numbers for the corresponding codes, respectively.

	implementation	refactoring	removing code	uncommenting code	removing workaround	other	not fixed
functionality needed	54	1	0	0	0	0	69
refactoring needed	2	16	1	0	0	1	29
clarification request	5	0	4	0	0	1	33
workaround	2	0	2	3	5	0	12
wait	2	0	2	3	0	0	6
bug	2	0	1	2	0	1	6
explanation	0	0	0	0	0	0	5
other	1	1	4	0	0	2	6

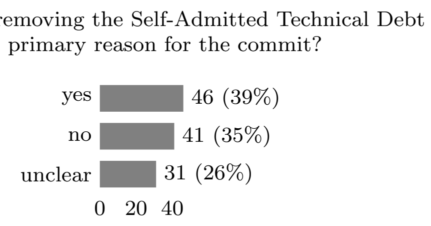

Fig. 6: Distribution of answers to “Was removing the Self-Admitted Technical Debt the primary reason for the commit?”. Agreement among the annotators: weighted kappa $\kappa = 0.630$ across 118 comments, i.e., “substantial” agreement (Viera and Garrett, 2005).

addressing the debt. An example is the self-admitted technical debt comment “`some OS such as Windows can have problem doing delete IO operations so we may need to retry a couple of times to let it work`” in the Apache Camel project which was removed in commit f10f55e[6] together with the surrounding source code. We hypothesise that in some cases, developers decide to replace code which requires an explanation with simpler code. More work will have to be conducted to test this hypothesis. Waits could sometimes be addressed by uncommenting code that had been written in anticipation of the fix. A large number of comments indicating debt were not addressed—for example, out of 43 comments which we coded as clarification request, 33 (77%) were “resolved” by simply deleting the comment (e.g., the comment “`TODO why zero?`” was removed from the Apache Camel source code in commit 3d8f4e9[7] without further explanation. Note that in cases where more than one of our codes could apply, we noted the most prominent one. This could for example occur in cases of long comments which were used to communicate different concerns. In such rare cases, we applied the code for the longest section of the comment. This explains the small number of inconsistencies, e.g., a “functionality needed” debt fixed by a “refactoring”.

3.5 RQ1.4 Is the removal of self-admitted technical debt the primary reason for the commits which remove the corresponding comments?

The removal of technical debt was often not the primary reason for commits which removed self-admitted debt, see Figure 6. We did not attempt to resolve disagreements between annotators for this question as the concept of “primary reason” can be ambiguous. Instead, instances where annotators disagreed are shown as “unclear” in Figure 6.

[6] `https://github.com/apache/camel/commit/f10f55e38945686827dc249703b1606682657a62`
[7] `https://github.com/apache/camel/commit/3d8f4e9d68253269b4f5cf7e3cfea4553b46d74f`

An example of a commit which removed self-admitted technical debt even though it was not the main purpose of the commit is Apache Camel commit f47adf.[8] The commit removed the following comment: "`TODO: Support ordering of interceptors`", but this was part of a much larger refactoring as described in the commit message: "`Overhaul of JMX`". On the other hand, the commit message of commit 88ca35[9] from the same project "`Added onException support to DefaultErrorHandler`" is very similar to the self-admitted technical debt comment that was removed in this commit "`TODO: in the future support onException`", which suggests that removing the debt was the primary reason for this commit.

3.6 RQ1.5 Could the fixes applied to address self-admitted technical debt be applied to address similar debt in other projects?

We annotated the 118 self-admitted technical debt comments which had been fixed by a commit in terms of whether the fix applied in this commit could be applied in a similar context in a different project. While this annotation was subjective to some extent—as also indicated by our kappa agreement of 0.540 which was the lowest across all questions we answered about the self-admitted technical debt comments—we used our intuition about whether we could envision tool support to address a comment automatically. We used our experience of conducting research on automated tool support for source code manipulation for this step.

We identified two kinds of self-admitted technical debt that could possibly be handled automatically. The first kind are comments which are fairly specific, e.g., "`TODO gotta catch RejectedExecutionException and properly handle it`". Automated tool support could be built to at least catch the exception based on this description. The second kind are comments which indicate that a developer is waiting for something, which we will discuss further in the next subsection. Figure 7 shows the ratio of fixes that could possibly be automated and applied in other settings, which is one third of all fixes. Note that we counted all those comments as "possibly" that were rated as "possibly" by at least one annotator. This finding supports Zampetti et al. (2018) who found that most changes addressing self-admitted technical debt require complex source code changes. The primary goal of investigating this research question was the identification of types of self-admitted technical debt likely amenable to being fixed automatically.

[8] `https://github.com/apache/camel/commit/f47adf75510ef71a5b4071e8c77af7abb9c07dc9`

[9] `https://github.com/apache/camel/commit/88ca359343c3a96786d435985f46841eeffcfb6e`

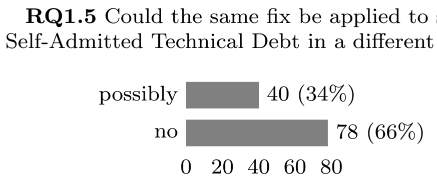

Fig. 7: Distribution of answers to “Could the same fix be applied to similar Self-Admitted Technical Debt in a different project?”. Agreement among the annotators: kappa $\kappa = 0.540$ across 118 comments, i.e., “moderate” agreement (Viera and Garrett, 2005).

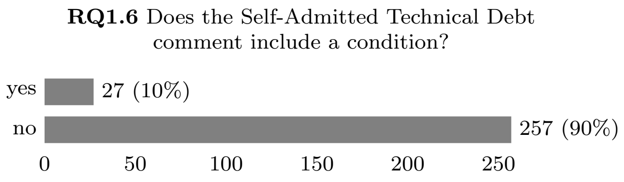

Fig. 8: Distribution of answers to “Does the Self-Admitted Technical Debt comment include a condition?”. Initial agreement among the annotators before resolving disagreements: weighted kappa $\kappa = 0.618$ across 284 comments, i.e., “substantial” agreement (Viera and Garrett, 2005).

Table 4: Example of self-admitted technical debt on “on-hold” and “wait”

Example of SATD	**Category / On-hold or not**
// TODO change to file when this is ready	wait / non on-hold
// FIXME: Code to be used in case route replacement is needed	wait / non on-hold
// TODO: is needed when we add support for when predicate	add functionality / on-hold
// TODO: Camel 2.9/3.0 consider moving to org.apache.camel	refactor / on-hold

3.7 RQ1.6 How many of the comments indicating self-admitted technical debt contain a condition to specify that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere?

A theme that emerged from answering the previous research question is the concept of self-admitted technical debt comments which include a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere. Since no other obvious class of self-admitted technical debt emerged which seemed amenable to automated tool support, we focus on this kind of self-admitted technical debt for building a classifier (see next section). We refer to this kind of debt as on-hold SATD—the comment is on hold until the condition is met (see Figure 1 for examples).

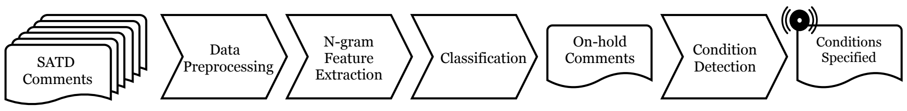

Fig. 9: Classification overview.

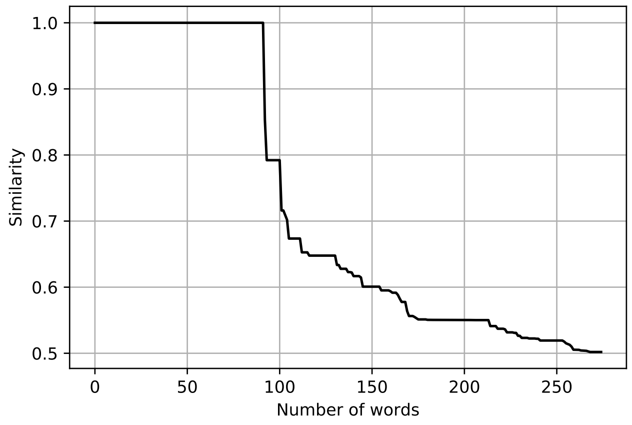

Fig. 10: Similarity between project names and words.

In our sample, we identified 27 such comments, see Figure 8. These comments are also related to the “wait” category shown in Figure 3, but not necessarily identical since the question addressed by Figure 3 did not explicitly ask about conditions. Table 4 shows examples of “on-hold” comments and those classified in the “wait” category.

4 Classifier Design

Figure 9 shows the overview of our classifier for on-hold SATD identification and the detection of the specific conditions that developers are waiting for. Given self-admitted technical debt comments, data preprocessing and n-gram feature extraction are applied before classifying them into on-hold or not. Within identified on-hold SATD comments, specific conditions are detected.

4.1 Data Preprocessing

Three preprocessing steps are applied, namely, term abstraction, lemmatization, and special character removal.

Table 5: Regular expressions for term abstraction

abstraction	pattern

@abstractdate	`(0[1-9]|[12]\d|3[01]).(0[1-9]|1[0-2])`
	`.([12]\d3)`
	# year.month.date, e.g., 21.02.2011
	`(0[1-9]|[12]\d|3[01])\/(0[1-9]|1[0-2])`
	`(\/([12]\d3))*`
	# day/month(/year), e.g., 25/05, 22/05/2012
	`((([0-9])|([0-2][0-9])|([3][0-1]))`
	`(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|`
	`Oct|Nov|Dec)\w+ \d4`
	# day month year, e.g., 23 June 2013
	`\d+-\d+-\d+ \d+:\d+:\d+ [-|+]\d+`
	# year-month-day timestamp, e.g., 2006-03-06 23:16:24 +0100

@abstractversion	`[0-9]{1,2}\.[0-9]{1,2}([+-]|\.[0-9]{1,3}|`
	`\.[A-Za-z]{1,2})*(_[0-9]{1,3})*`
	# release version, e.g., 1.9.3, 4.0, 8.0.x, 1.0.12_25

@abstractbugid	`abstractproduct[ |-]*\d+` # bug id, e.g., jetty-9.3

@abstracturl	`https?:\/\/(www\.)?[-a-zA-Z0-9@:%._\`
	`+~#=]{2,256}\.[a-z]{2,6}\b`
	`([-a-zA-Z0-9@:%_\+.~#?&//=]*)`
	# url

Term Abstraction. Similar to a previous text classification study (Prana et al., 2019), we perform abstraction as a preprocessing step. The previous study (Prana et al., 2019) abstracted keywords from GitHub README files. Their abstraction included mail-to links, hyperlinks, code blocks, images, and numbers. We also apply abstraction for hyperlinks (URLs), however, we do not apply the others because images, mail-to links, and code blocks do not usually appear in comments. Instead, we introduce four kinds of abstraction which are related to on-hold conditions. We target the following terms: *date expression*, *version*, *bug id*, *URL*, and *product name*. Each term is abstracted into a string: `@abstractdate`, `@abstractversion`, `@abstractbugid`, `@abstracturl`, and `@abstractproduct`. Table 5 shows the regular expressions we use to detect `@abstractdate`, `@abstractversion`, `@abstractbugid`, and `@abstracturl`.

For abstracting product names for `@abstractproduct`, we try finding semantically similar words to the project names and their sub-project names in our data set, i.e., Apache, Camel, Tomcat, Hadoop, Gerrit, Log4j, Yarn, Mapreduce, Hdfs, Ant, Jmeter, ArgoUML, Columba, Emf, Hibernate, Distribution, JEdit, JFreechart, JRuby, and SQuirrel. Figure 10 shows the similarity between each word in comments and project name and their related project using Spacy (Honnibal and Montani, 2017).[10] According to the result, the similarity score drops drastically from 1.0—therefore, we consider words with similarity 1.0 as project names. We obtained 77 words.

We apply this process because we are more interested in the existence of these types rather than the actual terms, which do not appear frequently. For example, considering the comment “`TODO: CAMEL-1475 should fix this`”, CAMEL-1475 will be changed to the string “`@abstractproduct @abstractbugid`”. Table 5 summarizes the regular expressions we used for identifying targeted terms. Replacements using the regular expressions are conducted from top to bottom in the table. Subsequently, URLs linking to specific ids of bugs are abstracted to “`@abstracturl @abstractbugid`”.

Lemmatization. Lemmatization is a process to reduce the inflection form of words into dictionary form by considering the context in the sentences. This process is applied to increase the frequency of words appearing by changing words into dictionary forms using tools from Spacy (Honnibal and Montani, 2017).

Special character removal. Since non-English characters and non-numeric ones do not represent words, we use the regular expression `[^A-Za-z0-9]+` to remove them. Stop word removal is not applied in this work because a stop word list contains important keywords for identifying on-hold SATD (e.g., when, until). We use Spacy to apply lemmatization which will change words into their dictionary form. However, some single characters will appear, e.g., when lemmatising “// TODO: Removed from UML 2.x” to “todo remove from uml 2 x”.

4.2 N-gram Feature Extraction

We extract n-gram term features by applying N-gram IDF (Shirakawa et al., 2015, 2017). Inverse Document Frequency (IDF) has been widely used in many applications because of its simplicity and robustness; however, IDF cannot handle phrases (i.e., groups of more than one term). Because IDF gives more weight to terms occurring in fewer documents, rare phrases are assigned more weight than good phrases that would be useful in text classification. N-gram IDF is a theoretical extension of IDF for handing multiple terms and phrases by bridging the theoretical gap between term weighting and multi-word expression extraction (Shirakawa et al., 2015, 2017).

Terdchanakul et al. (2017) reported that for classifying bug reports into bugs or non-bugs, classification models using features from N-gram IDF outperform models using topic modeling features. In addition to this, we consider

[10] Spacy has recently been found to achieve a higher accuracy when applied to software-related text compared to other libraries (Al Omran and Treude, 2017).

that n-gram word features are beneficial for comment classification rather than topic modeling because source code comments are generally short and contain only a small number of words.

Wattanakriengkrai et al. (2018) created classification models to identify design and requirement self-admitted technical debt using source code comments. By using N-gram IDF and auto-sklearn automated machine learning, classification models outperform models with single word features.

In this study, we use an N-gram Weighting Scheme tool (Shirakawa, 2017), which uses an enhanced suffix array (Abouelhoda et al., 2004) to enumerate valid n-grams. We obtain a list of all valid n-grams that contain at most 10 terms from the on-hold self-admitted technical debt comments and remove n-grams which have frequency equal to one. We obtain about one thousand two hundred n-gram terms from our 267 on-hold self-admitted technical debt comments. After that, we apply Auto-sklearn's feature selection. Auto-sklearn includes two feature selection functions from the sklearn library, sklearn.feature_selection.GenericUnivariateSelect (Univariate feature selector) and sklearn.feature_selection.SelectPercentile (Select features according to percentile). Calling these functions is part of Auto-sklearn's feature preprocessing—it selects suitable feature processing based on meta-learning automatically.

4.3 Classifier Learning

Given the set of n-gram term features from the previous step, we build a classifier that can identify on-hold SATD by classifying self-admitted technical debt comments into on-hold or not.

In machine learning, two problems are known: (1) no single machine learning method performs best on all data sets, and (2) some machine learning methods rely heavily on hyperparameter optimization. Automated machine learning aims to optimize choosing a good algorithm and feature preprocessing steps (Feurer et al., 2015). To obtain the best performance (RQ2.1), similar to Wattanakriengkrai et al. (2018)'s work, we apply auto-sklearn (Feurer et al., 2015), a tool of automated machine learning.

Auto-sklearn addresses these problems as a joint optimization problem (Feurer et al., 2015). Auto-sklearn includes 15 base classification algorithms, and produces results from an ensemble of classifiers derived by Bayesian optimization (Feurer et al., 2015).

For classifier learning, we prepare feature vectors with N-gram TF-IDF scores of all n-gram terms. The score is calculated with the following formula:

$$\textit{n-gram TF-IDF} = log(\frac{|D|}{sdf}) * gtf$$

where $|D|$ is the total number of comments, sdf is the document frequency of a set of terms composing an n-gram, and gtf is the global term frequency.

4.4 On-hold Condition Detection

After on-hold SATD comments are identified, we try to identify their on-hold conditions. During our annotation, we found conditions of self-admitted technical debt that are related to waiting for a bug to be fixed, a release of a library, or a new version of a library.

- For a bug to be fixed, we abstract the bug report number. In a bug report tracking system, the bug report number is created by using the project name and report number which we abstract using the keywords @abstractproduct and @abstractbugid.
- For release date, we abstract it using the keyword @abstractdate.
- For a new version of a library, the library version usually appears in a project name and release version (e.g., 1.9.3, 4.0), which we abstract using the keywords @abstractproduct and @abstractversion.

As we have already replaced these terms with specific keywords shown in Table 5, we can derive conditions by recovering the original terms. The following is our detection process.

1. Extract keywords of `@abstractdate`, `@abstractversion`, `@abstractbugid`, and `@abstractproduct` by preserving the order of appearance in the identified on-hold SATD comments.
2. Group keywords to make valid conditions. Only the following sets of keywords are considered to be valid conditions, and other keywords that do not match the following orders are ignored.
   - {`@abstractdate`}: an individual date expression.
   - {`@abstractproduct, @abstractversion, ...`}: a product name followed by one or more version expressions, to indicate specific versions of the product.
   - {`@abstractproduct, @abstractbugid, ...`}: a product name followed by one or more bug ID expressions, to indicate specific bugs of the product.

Identifying these keywords as conditions is not trivial, because they also frequently appear in comments that do not indicate on-hold SATD. Since we limit this detection to the identified on-hold comments, we expect that this simple process can work.

5 Classifier Evaluation

In this section, we describe the steps we took to evaluate our classifier.

5.1 Data Preparation and Annotation

As shown in Figure 8, we found fewer than 30 on-hold SATD comments in the sample of 333 comments. Since it is difficult to train classifiers on such a small

Table 6: Annotated self-admitted technical debt comments

	characteristic	**number**
excluded	not self-admitted technical debt	225
	sample of removed self-admitted technical debt	333
classification data	with condition (on-hold)	267
	without condition	4,981
sum		5,806

number of instances, we investigated all 2,507 comments again to prepare data for our classification. After that, the first and third author separately annotated the remaining comments in terms of (i) whether comments represent self-admitted technical debt (similar to Figure 2) and (ii) whether the self-admitted technical debt comments include a condition (similar to Figure 8). All conflicts in this annotation were resolved by the second author. Note that we decided to train the classifier on comments which had been removed through the resolution of self-admitted technical debt to ensure we were able to consider the entire lifecycle of the self-admitted technical comment before deciding whether to consider it on-hold.

We also include a data set from ten open source projects introduced by Maldonado et al. (2017b). First, we randomly selected a sample of 30 comments out of all 3,299 comments. The first author, third author, and an external annotator annotated these comments, resulting in 97.78% overall agreement, i.e., "almost perfect" according to Viera and Garrett (2005). Then the first author annotated the remaining comments.

Tables 6 and 7 show the result of this data preparation. From 5,806 comments, 333 comments are samples of removed self-admitted technical debt and 225 comments that do not represent self-admitted technical debt are excluded. We obtained 267 on-hold comments and 4,981 other comments, which are used for our classification. After excluding duplicate comments, our dataset contains a total of 5,248 comments. Before exclusion, 410 comments were duplicates which can be grouped into 168 sets of comments. Among these 168 sets of duplicates, 11 sets (24 comments) are on-hold comments, and 157 sets (386 comments) are non on-hold comments.

5.2 Evaluation Settings

We measure the classification performance in terms of precision, recall, F_1, and AUC. AUC is the area under the receiver operating characteristic curve. The receiver operating characteristic curve is created by plotting the true positive rate (TPR) against the false positive rate (FPR) at various threshold settings.

$$Precision = \frac{tp}{tp + fp}$$

Table 7: Number of on-hold SATD comments in each project

project	**number**	**example**
Apache Camel	88	// @deprecated will be removed on Camel 2.0 ...
Apache Tomcat	20	// TODO This can be fixed in Java 6 ...
Apache Hadoop	20	// TODO need to get the real port number MAPREDUCE-2666
Gerrit Code Review	6	// TODO: remove this code when Guice fixes its issue 745
Apache Log4j	1	// TODO: this method should be removed if OptionConverter becomes a static
Apache Ant	7	// since Java 1.4 ... // workaround for Java 1.2-1.3
Apache Jmeter	2	// TODO this bit of code needs to be tidied up ... Bug 47165
ArgoUML	77	// TODO: gone in UML 2.1
Columba	0	–
EMF	1	// Note: Registry based authority is being removed ... which would obsolete RFC 2396. If the spec is added ... needs to be removed.
Hibernate	5	// FIXME Hacky workaround to JBCACHE-1202
JEdit	6	// undocumented hack to allow browser actions to work. // XXX - clean up in 4.3
JFreeChart	2	// TODO: In JFreeChart 1.2.0 ...
JRuby	23	// Workaround for JRUBY-4149
SQuirrel	9	// We know this fails - Bug# 1700093
total	267	–

$$Recall = \frac{tp}{tp + fn}$$

$$F_1 = \frac{2 \cdot (precision \cdot recall)}{(precision + recall)}$$

$$TPR = \frac{tp}{tp + fn}$$

$$FPR = \frac{fp}{tn + fp}$$

where tp is the number of true positives, tn is the number of true negatives, fp is the number of false positives, and fn is the number of false negatives.

Comparison. A Naive Baseline is created based on the assumption that it is also possible to find on-hold technical debt comments while using basic searching similar to the `grep` command. The words we use for searching are selected from the top 30 words that appear frequently in comments. We manually classify words to select those that relate to on-hold technical debt. The words we selected are *"should"*, *"when"*, *"once"*, *"remove"*, *"workaround"*, *"fixed"*, *"after"*, and *"will"*.

To assess the effectiveness of n-gram features in classifying on-hold SATD comments, we compare the performances of classifiers using N-gram TF-IDF

Table 8: Performance comparison

	naive baseline	**TF-IDF**	**N-gram TF-IDF**	**N-gram TF-IDF without rebalancing**
Precision	0.12	0.73	0.75	**0.76**
Recall	0.66	0.60	**0.78**	0.77
F_1	0.20	0.66	**0.77**	0.76
AUC	0.70	0.97	**0.98**	**0.98**

Table 9: Top 10 N-gram TF-IDF frequent features only appear in on-hold comments.

N-gram Features	**frequency**
‘remove’, ‘in’, ‘abstractproduct’, ‘abstractversion’	7
‘in’, ‘uml’, ’2’, ‘x’	7
‘fix’, ‘in’	6
‘workaround’, ‘to’	6
‘todo’, ‘cmueller’, ‘remove’, ‘the’	6
‘ref’, ‘attribute’	6
‘be’, ‘remove’, ‘in’, ‘abstractproduct’, ‘abstractversion’	6
‘for’, ‘abstractversion’	5
‘after’, ‘abstractproduct’, ‘abstractbugid’	5
‘workaround’, ‘for’, ‘abstractproduct’, ‘abstractbugid’	4

and traditional TF-IDF (Salton and Buckley, 1988). Except for feature extraction, the two classifiers are prepared using the same settings including term abstraction.

Ten-fold cross-validation. Ten-fold cross-validation divides the data into ten sets and every set is used as test set once while the others are used for training. Due to the imbalance between the number of positive and negative instances, we use the Stratified ShuffleSplit cross validator of scikit-learn made available by Pedregosa et al. (2011), which intends to preserve the percentage of samples from each class. Because of this process, some instances can appear multiple times in different sets. Therefore we report the mean values of the evaluation metrics across all ten runs as the performance.

To measure the effect of rebalancing on our classification, we compare the performance of N-gram TF-IDF with and without Stratified ShuffleSplit.

5.3 RQ2.1 What is the best performance of a classifier to automatically identify on-hold SATD?

As shown in Table 8, our classifier with n-gram TF-IDF achieved a mean precision of 0.75, a mean recall of 0.78, a mean F_1-score of 0.77, and a mean AUC of 0.98. N-gram TF-IDF has the best performance in every evaluation except precision which has a similar score with N-gram TF-IDF without rebalancing. We consider that both precision and recall are essential for this kind

Table 10: Cross-project classification on projects which contain on-hold more than 2%

Project	**#, (% of on-hold)**	**Precision**	**Recall**	**F1**	**AUC**
Apache Ant	7, (5.6%)	0.50	0.57	0.53	0.97
Apache Camel	88, (10.9%)	0.81	0.39	0.52	0.96
Apache Hadoop	20, (8.2%)	0.61	0.85	0.71	0.96
Apache Tomcat	20, (2.8%)	0.19	0.50	0.27	0.92
ArgoUML	77, (6.7%)	0.36	0.17	0.23	0.82
Gerrit Code Review	6, (6.3%)	0.60	0.50	0.55	0.91
JEdit	6, (2.6%)	0.33	0.67	0.44	0.91
JRuby	23, (5.0%)	0.46	0.57	0.51	0.97
SQuirrel	9, (3.9%)	0.24	0.44	0.31	0.91
Average	-	0.46	0.52	0.45	0.93

Table 11: Within-project classification on projects which contain on-hold more than 2%

Project	**#, (% of on-hold)**	**Precision**	**Recall**	**F1**	**AUC**
Apache Ant	7, (5.6%)	0.19	0.80	0.30	0.86
Apache Camel	88, (10.9%)	0.91	0.61	0.73	0.99
Apache Hadoop	20, (8.2%)	0.85	0.80	0.79	0.99
Apache Tomcat	20, (2.8%)	0.73	0.65	0.66	0.99
ArgoUML	77, (6.7%)	0.66	0.79	0.71	0.98
Gerrit Code Review	6, (6.3%)	0.15	0.30	0.20	0.86
JEdit	6, (2.6%)	0.11	0.80	0.18	0.90
JRuby	23, (5.0%)	0.70	0.80	0.70	0.99
SQuirrel	9, (3.9%)	0.15	1.00	0.25	0.89
Average	-	0.49	0.73	0.50	0.94

of recommendation system. Precision is important since false positives (i.e., unwarranted recommendations) will annoy developers. However, recall is still important since false negatives (i.e., recommendations that the system could have made but did not) might cause problems since developer will be unaware of important information. Table 9 shows the top features from N-gram TF-IDF ranked by how frequently our classifier uses them to distinguish on-hold comments from other self-admitted technical debt comments.

We also run an experiment for both cross-project classification and within-project classification on projects for which the ratio of on-hold SATD comments among all self-admitted technical debt comments is more than 2%. For cross-project classification, we divide data into sets according to their project. Every set is used as test set once while the other sets are used for training. Table 10 shows the results for each project. On average, our classifier with cross-project classification achieved a mean precision of 0.46, a mean recall of 0.52, a mean F_1-score of 0.45, and a mean AUC of 0.93.

For within-project classification, in each project, we apply a ten-fold classification with the Stratified ShuffleSplit cross validator. Table 11 shows the

Table 12: Examples of specific conditions in on-hold SATD comments

specific condition	**example of on-hold SATD comments**
@abstractdate	// Workaround for, Adobe Read 9 plug-in on IE bug // Can be removed after **26 June 2013**
@abstractproduct, @abstractversion	// TODO cmueller:, remove the "httpBindingRef" look up in **Camel 3.0**
@abstractproduct, @abstractbugid	// FIXME (**CAMEL-3091**): @Test

```
/*
* TODO: After YARN-2 is committed, we should call containerResource.getCpus()
* (or equivalent) to multiply the weight by the number of requested cpus.
*/
```

Fig. 11: On-hold SATD example which we correctly identify[11]

result for each project. Among them, five projects (Camel, Hadoop, Tomcat, ArgoUML, and JRuby) have a similar score or higher compared to cross-project classification according to all metrics. Another four projects (Ant, Gerrit, JEdit, and SQuirrel) have a lower score compared to cross-project classification according to all metrics. The difference between these two groups is that the first group has the number of on-hold comments >= 20 while the second group has the number of on-hold comments < 10.

5.4 RQ2.2 How well can our classifier automatically identify the specific conditions in on-hold SATD?

Because of our treatment of imbalanced data (see Section 5.3), some comments can appear multiple times in the test set. We consider that an on-hold comment is correctly identified only if it has been classified correctly in all cases where it was part of the test set. Our classifier was able to identify 219 out of 267 on-hold comments correctly. Among them, 94 comments contain abstraction keywords which indicate a specific condition, and all those instances were confirmed to be specific conditions by manual investigation. Table 12 shows examples of on-hold comments and their specific conditions. However, some comments do not mention specific conditions, such as "`This crap is required to work around a bug in hibernate`". Among the 48 false positives (incorrectly identified comments), 10 comments contain abstraction keywords, but these keywords are used for references and not for conditions that a developer is waiting for. In summary, 90% (94/(10+94)) of the detected specific conditions are correct, and for 43% (94/219) of the on-hold comments, we were able to identify the specific condition that a developer was waiting for.

[11] cf. `https://github.com/apache/hadoop/commit/80eb92aff02cc9f899a6897e9cbc2bc69bd56136/hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server`

```
/**
 * Ugly workaround because CodeMirror never hides lines completely.
 * TODO: Change to use CodeMirror's official workaround after
 * updating the library to latest HEAD.
 */
```

Fig. 12: On-hold SATD example which our classifier cannot identify[12]

Figure 11 shows an example of an on-hold SATD comment. Our model can identify conditions using the keywords @abstractproduct and @abstractbugid referring to YARN-2. Figure 12 shows an example that our classifier could not identify correctly. The on-hold condition refers to a workaround waiting for an update to the CodeMirror library.

5.5 Developer Feedback

To evaluate whether our approach for detecting on-hold SATD could be useful in practice, we ran the cross-project classifier on the source code of the open-source project JabRef, a graphical Java application for managing BibTeX and biblatex (.bib) databases.[13] We used source code comments containing SATD keywords from Huang et al. (2018) (Table 1 and Table 16) and an abstractkeyword as input and classified the resulting data into on-hold and not on-hold. From the classification result, we obtained a total of 22 potential on-hold comments. A manual analysis revealed that 19 cases were not actually SATD and that 3 cases are on-hold SATD. Note that our classifier was developed to classify SATD comments into on-hold or not, and not to determine whether any comment is SATD—this has been done in previous work (Maldonado and Shihab (2015)). We then sent three instances of on-hold SATD to one of JabRef's core developers. Table 13 shows these comments along with the explanation we sent to the developer. In addition, for each on-hold SATD comment, we included a link to the exact line of code from which we had extracted the comment.

Regarding the first comment, the developer pointed out that the comment had been deleted in the meantime, but noted

As a final check, it would be helpful though.

Regarding the second comment, the developer mentioned the potential overlap between notifications that our approach could produce and other notifications that the developer would already receive anyway:

-nodemanager/src/main/java/org/apache/hadoop/yarn/server/nodemanager/util/CgroupsLCEResourcesHandler.java

[12] cf. https://github.com/gerrit-review/gerrit/commit/0485172aaa70e3b1f0e98c00215672657e6f462e/gerrit-gwtui/src/main/java/com/google/gerrit/client/diff/CodeMirrorDemo.java

[13] https://github.com/JabRef/jabref/

Table 13: On-hold SATD sent for developer feedback

#	on-hold SATD	explanation
1	"... todo: reenable customize entry types feature (<link to issue 4719>) ..."	we could notify developers once issue 4719 has been closed
2	"... we must not clean the url. this is the deal with @manastungare - see <link to comment on issue 684> ..."	we could have notified developers once issue 684 was closed and/or if there have been responses to the comment
3	"... - handling of identically fields with different names (<link to issue 521>) ..."	we could have notified developers once issue 521 was closed

Maybe, here an active 'Comment Checking' would be more helpful. Then remembering if the comment should be kept - so that an additional scan with the same setting does not trigger a notification again. - For me, getting notified because of new comments additionally, would not be helpful as I would have been notified of GitHub.

The third comment turned out to be the most useful one, in the developer's perception:

In this case, a bot posting a message to the issue with following text would have been helpful: 'I found following references to this closed issue in the code. Maybe, the code has to be adapted, too?'

In response to our final question "Do you think such a tool could be useful?", the developer responded

Since the last example was really useful, you hear me saying: "Yes". :-).

6 Implications

The ultimate goal of our work is to enable the automated management of certain kinds of self-admitted technical debt. Previous work (Zampetti et al., 2018) has found that most changes which address self-admitted technical debt require complex code changes—as such, it is unrealistic to assume that automated tool support could handle all kinds of requirement debt and design debt that developers admit in source code comments. Thus, in this work we set out to first identify a sub-class of self-admitted technical debt amenable to automated management and second develop a classifier which can reliably identify this sub-class of debt.

Our qualitative study revealed one particular class of self-admitted technical debt potentially amendable to automated tooling: on-hold SATD, i.e., comments in which developers express that they are waiting for a certain external event or updated functionality from an external library before they can address the debt that is expressed in the comment. In other words, the comment is on hold until the condition has been met.

Based on the data set made available by Maldonado et al. (2017a) and Maldonado et al. (2017b), we identified a total of 293 comments which indicate on-hold SATD, confirming that this phenomenon is prevalent and exists in different projects. Our classifier to identify on-hold SATD was able to reach an AUC of 0.98 in identifying comments that belong to this sub-class. In addition, we were able to identify specific conditions contained within these comments (90% of conditions are detected correctly). Based on 15 projects, there are 293 on-hold comments out of 5,529 self-admitted technical debt comments, resulting in a relative frequency of 5.30%. Out of 15 projects, the ratio of on-hold comments compared to all self-admitted technical debt comments is larger than 2% for nine projects.

Given all the events and new releases that happen in a software project at any given point in time, it is unrealistic to assume that developers will be able to stay on top of all instances of technical debt that are ready to be addressed once a condition has been met. Instead, there is a risk that developers forget to go back to these comments and debt instances even when the event they were originally waiting for has occurred.

This work builds a first step towards the design of automated tools that can support developers in addressing certain kinds of self-admitted technical debt. In particular, based on the classifier introduced in this work, it is now possible to build tool support which can monitor the specific external events we have identified in this work (e.g., certain bug fixes or the release of new versions of external libraries) and notify developers as soon as a particular debt is ready to be addressed. While the ratio of on-hold comments is fairly low, such comments appeared in almost all of the studied projects, and we argue that alerting developers when such comments are ready to be addressed can prevent bugs or vulnerabilities that might otherwise occur, e.g., because of outdated libraries.

In terms of tool support, we envision a tool which supports the developer by indicating comments that are ready to be addressed rather than a tool which addresses comments automatically. Addressing comments automatically—even though it is an interesting research challenge—is problematic for two reasons: (1) the precision of such a tool would have to be really high, and current work including our own suggests that this is not yet the case; and (2) developers are unlikely to relinquish control over their code base to a tool which automatically changes code.

7 Threats to Validity

Regarding threats to *internal validity*, it is possible that we introduced bias through our manual annotation. While we generally achieved high agreement regarding the annotation questions listed in Table 2, the initial agreement regarding RQ1.1 was low which is explained by the nature of the open-ended question. We resolved all disagreements through multiple co-located coding sessions with the first three authors of this paper. Note that we do not use the results of RQ1.1 as an input for our classifier. We may possibly have wrongly classified the removal of self-admitted technical debt, since in particular for comments indicating the need for new features, it can be hard to judge whether the new feature was indeed fully implemented. Another concern is that we did not manually validate the entire data set that we are reusing from previous work. There might be further quality issues with this which would affect the performance of our approach, such as self-admitted technical debt being identified in the header of classes.

For *external validity*, while we analyzed a statistically representative sample of commits for RQ1 and the entire data set made available by Maldonado et al. (2017a) (after removing duplicates) for RQ2, we cannot claim generalizablity beyond the projects contained in this data set and our classifier might be biased as a result of the small number of projects. The limited data set allowed us to perform an in-depth qualitative analysis, and future work will need to investigate the applicability of our results to other projects and within-project prediction.

For *construct validity*, this is related to the manual labeling of on-hold SATD. A label might be affected by annotator misunderstand or mislabeling. Despite annotators resolving disagreements through discussion, the labels might still be incorrect.

8 Related Work

Self-admitted technical debt has been a popular research topic in the software engineering community in recent years. In this section, we introduce key research related to our study.

8.1 Impact of self-admitted technical debt

Sierra et al. (2019) conducted a survey about self-admitted technical debt by investigating three categories: (i) detection, (ii) comprehension, and (iii) repayment. Detection focuses on identifying and detecting self-admitted technical debt. Comprehension studies the life cycle of self-admitted technical debt. Repayment focuses on removal of self-admitted technical debt. This research found a lack of research related to the repayment of self-admitted technical debt.

Maldonado et al. (2017a) studied the removal of self-admitted technical debt by applying natural language processing to self-admitted technical debt. They found that (i) the majority of self-admitted technical debt was removed, (ii) self-admitted technical debt was often removed by the person who introduced it, and (iii) self-admitted technical debt lasts between 18 to 172 days (median). Using a survey, the authors also found that developers mostly use self-admitted technical debt to track bugs and code that requires improvement. Developers mostly remove self-admitted technical debt when they are fixing bugs or adding new features.

Zampetti et al. (2018) conducted an in-depth quantitative and qualitative study of self-admitted technical debt. They found that (i) 20% to 50% of the corresponding comments were accidentally removed when entire methods or classes were dropped, (ii) 8% of self-admitted technical debt removals were indicated in the commit messages, and (iii) most of the self-admitted technical debt requires complex changes, often changing method calls or conditionals.

Bavota and Russo (2016) introduced a large-scale empirical study across 159 software projects. From this data they performed manual analysis of 366 comments, showing (i) an average of 51 self-admitted technical debt comments per system, (ii) that self-admitted technical debt consists of 30% code debt, 20% defect debt, and 20% requirement debt, (iii) the number of self-admitted technical debt comments is increasing over time, and (iv) on average it takes over 1,000 commits before self-admitted technical debt is fixed.

Wehaibi et al. (2016) studied the relation between self-admitted technical debt and software quality based on five open source projects (i.e., Hadoop, Chromium, Cassandra, Spark, and Tomcat). Their result showed that (i) there is no clear evidence that files with self-admitted technical debt had more defects than other files, (ii) compared with self-admitted technical debt changes, non-debt changes had a higher chance of introducing other debt, but (iii) changes related to self-admitted technical debt were more difficult to achieve.

Mensah et al. (2018) introduced a prioritization scheme. After running this scheme on four open source projects, they found four causes of self-admitted technical debt which was code smells (23.2%), complicated and complex task (22.0%), inadequate code testing (21.2%), and unexpected code performance (17.4%). The result also showed that self-admitted technical design debt was prone to software bugs, and that for highly prioritized self-admitted technical debt tasks, more than ten lines of code were required to address the debt.

Kamei et al. (2016) used analytics to quantify the interest of self-admitted technical debt to see how much of the technical debt incurs positive interest, i.e., debt that indeed costs more to pay off in the future. They found that approximately 42–44% of the technical debt in their case study incurred positive interest.

Palomba et al. (2017) conducted an exploratory study on the relationship between changes and refactoring and found that developers tend to apply a higher number of refactoring operations aimed at improving maintainability and comprehensibility of the source code when fixing bugs. In contrast, when new features are implemented, more complex refactoring operations are performed to improve code cohesion. In most cases, the underlying reasons behind the application of such refactoring operations were the presence of duplicate code or previously introduced self-admitted technical debt.

Mensah et al. (2016) propose a new technique to estimate Rework Effort, i.e., the effort involved to resolve self-admitted technical debt. They performed an exploratory study using text mining to extract self-admitted technical debt from source code comments. In order to extract source code comments, the authors apply text mining on four open source projects. The result from four projects shows a rework effort between 13 and 32 commented lines of code on average per self-admitted technical debt comment.

8.2 Self-admitted technical debt Identification and Classification

Potdar and Shihab (2014) tried to identify self-admitted technical debt by looking into source-code comments in four open source project (i.e., Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML). Their study showed that (i) the amount of debt in these project ranged between 2.4% and 31% of all files, (ii) debt was created mostly by developers with more experience, and time pressures and code complexity did not correlate with the amount of self-admitted technical debt, and (iii) only 26.3% to 63.5% of self-admitted technical debt comments were removed.

Farias et al. (2015) proposed a tool called CVM-TD (Contextualized Vocabulary Model for identifying Technical Debt) to identify technical debt by analyzing code comments. The authors performed an exploratory study on two open source projects. The result indicated that (1) developers use dimensions of CVM-TD when writing code comments, (2) CVM-TD provides vocabulary that may be used to detect technical debt, and (3) models need to be calibrated.

Farias et al. (2016) investigated the use of CVM-TD with the purpose of characterizing factors that affect the accuracy of the identification of technical debt, and the most chosen patterns by participants as decisive to indicate technical debt items. The authors conducted a controlled experiment to evaluate CVM-TD, considering factors such as English skills and experience of developers.

Silva et al. (2016) investigated the identification of technical debt in pull requests. The authors found that the most common technical debt categories are design, test, and project convention.

Maldonado et al. (2017b) tried identifying design-related and requirement-related self-admitted technical debt using a maximum entropy classifier.

Huang et al. (2018) tried classifying comments in terms of whether they contained self-admitted technical debt or not, and reported that their proposal outperformed the baseline method.

Maldonado and Shihab (2015) studied types of self-admitted technical debt using source code comments. This study classified types of self-admitted technical debt into design debt, defect debt, documentation debt, requirement debt, and test debt. The most common type of self-admitted technical debt is design debt and the second most common type is requirement debt. Self-admitted technical debt consist of 42% to 84% design debt, and 5% to 45% requirement debt.

Zampetti et al. (2017) developed a machine learning approach to recommend when design technical debt should be self-admitted. They found their approach to achieve an average precision of about 50% and a recall of 52%. When predicting cross-projects, the performance of the approach improved to an average precision of 67% and a recall of 55%.

Yan et al. (2019) identify self-admitted technical debt using change-level self-admitted technical debt determination. This model identifies whether a change introduces self-admitted technical debt. In order to create the model, they identified technical debt using all versions of source code comments. Then, they manually label changes that introduce technical debt in comments and extract 25 features which belong to three groups, i.e., diffusion, history, and message. After that, they create a classifier using random forest. Across seven projects, this model achieves an AUC of 0.82 and cost-effectiveness of 0.80.

Flisar and Podgorelec (2019) developed a new method to detect self-admitted technical debt using word embedding trained from unlabeled code comments. They then apply feature selection methods (Chi-square, Information Gain, and Mutual Information), and use three classification algorithms (Naive Bayes, Support Vector Machine, and Maximum Entropy) to test on ten open source projects. Their proposed method was able to achieve 82% correct predictions.

Liu et al. (2018) proposed a self-admitted technical debt detector tool which is able to detect debt comments using text mining and is able to manage detected comments in an IDE via an Eclipse plug-in.

Ren et al. (2019) proposed a Convolutional Neural Network for classifying code comments as self-admitted technical debt or not, based on ten open source projects. Their approach outperforms text-mining-based methods both in terms of within-project and cross-project prediction.

In our study we use the same data set as previous research (Maldonado et al., 2017a,b).

9 Conclusions and Future Work

Self-admitted technical debt refers to situations in which software developers explicitly admit to introducing technical debt in source code comments, arguably to make sure that this debt is not forgotten and that somebody will be able to go back later to address this debt. In this work, we hypothesize that it is possible to develop automated techniques to manage a subset of self-admitted technical debt.

As a first step towards automating a part of the management of certain kinds of self-admitted technical debt, in this paper, we contribute (i) a qualitative study on the removal of self-admitted technical debt in which we annotated a statistically representative sample of 333 technical debt comments using seven questions that emerged as part of the qualitative analysis; (ii) on-hold SATD (debt which contains a condition to indicate that a developer is waiting for a certain event or an updated functionality having been implemented elsewhere) which emerged from this qualitative analysis as a particular class of self-admitted technical debt that can potentially be managed automatically;

and (iii) the design and evaluation of a classifier for self-admitted technical debt which can detect on-hold debt with an AUC of 0.98 as well as identify the specific conditions that developers are waiting for.

Building on these contributions, in our future work we intend to build the tool support that our classifier enables: a recommender system which can indicate for a subset of self-admitted technical debt in a project when it is ready to be addressed. We found that self-admitted technical debt is sometimes addressed by uncommenting source code that has already been written in anticipation of the debt removal. As another step towards the automation of technical debt removal, in future work, we will explore whether it is possible to address such debt automatically.

References

Abouelhoda MI, Kurtz S, Ohlebusch E (2004) Replacing suffix trees with enhanced suffix arrays. Journal of Discrete Algorithms 2(1):53–86

Al Omran FNA, Treude C (2017) Choosing an nlp library for analyzing software documentation: A systematic literature review and a series of experiments. In: Proceedings of the International Conference on Mining Software Repositories, pp 187–197

Bavota G, Russo B (2016) A large-scale empirical study on self-admitted technical debt. In: Proceedings of the Working Conference on Mining Software Repositories, pp 315–326

Bazrafshan S, Koschke R (2013) An empirical study of clone removals. In: Proceedings of the International Conference on Software Maintenance, pp 50–59

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 Joint Meeting on Foundations of Software Engineering, pp 50–60

Feurer M, Klein A, Eggensperger K, Springenberg J, Blum M, Hutter F (2015) Efficient and robust automated machine learning. In: Cortes C, Lawrence ND, Lee DD, Sugiyama M, Garnett R (eds) Advances in Neural Information Processing Systems 28, Curran Associates, Inc., pp 2962–2970

Honnibal M, Montani I (2017) spacy - industrial-strength natural language processing in python. `https://spacy.io/`, (Accessed on 13/04/2019)

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

Ichinose T, Uemura K, Tanaka D, Hata H, Iida H, Matsumoto K (2016) ROCAT on KATARIBE: Code visualization for communities. In: Proceedings of the International Conference on Applied Computing and Information Technology, pp 158–163

Kamei Y, Maldonado E, Shihab E, Ubayashi N (2016) Using analytics to quantify the interest of self-admitted technical debt. CEUR Workshop Proceedings 1771:68–71

Kniberg H (2013) Good and bad technical debt (and how TDD helps). http://blog.crisp.se/2013/10/11/henrikkniberg/good-and-bad-technical-debt

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

Maldonado E, Shihab E (2015) Detecting and quantifying different types of self-admitted technical debt. In: Proceedings of the International Workshop on Managing Technical Debt, pp 9–15

Maldonado E, Abdalkareem R, Shihab E, Serebrenik A (2017a) An empirical study on the removal of self-admitted technical debt. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp 238–248

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

McConnell S (2007) Technical debt. http://www.construx.com/10x_Software_Development/Technical_Debt/

Mensah S, Keung J, Bosu M, Bennin K (2016) Rework effort estimation of self-admitted technical debt. CEUR Workshop Proceedings 1771:72–75

Mensah S, Keung J, Svajlenko J, Bennin KE, Mi Q (2018) On the value of a prioritization scheme for resolving self-admitted technical debt. Journal of Systems and Software 135(C):37–54

Palomba F, Zaidman A, Oliveto R, De Lucia A (2017) An exploratory study on the relationship between changes and refactoring. In: Proceedings of the International Conference on Program Comprehension, pp 176–185

Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay E (2011) Scikit-learn: Machine learning in Python. Journal of Machine Learning Research 12:2825–2830

Potdar A, Shihab E (2014) An exploratory study on self-admitted technical debt. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp 91–100

Prana GAA, Treude C, Thung F, Atapattu T, Lo D (2019) Categorizing the content of github readme files. Empirical Software Engineering

Salton G, Buckley C (1988) Term-weighting approaches in automatic text retrieval. Information Processing and Management 24(5):513–523

Shirakawa M (2017) Github - iwnsew/ngweight: N-gram weighting scheme. https://github.com/iwnsew/ngweight, (Accessed on 04/13/2019)

Shirakawa M, Hara T, Nishio S (2015) N-gram idf: A global term weighting scheme based on information distance. In: Proceedings of the International Conference on World Wide Web, pp 960–970

Shirakawa M, Hara T, Nishio S (2017) Idf for word n-grams. ACM Transactions on Information Systems 36(1):5:1–5:38

Shull FJ, Carver JC, Vegas S, Juristo N (2008) The role of replications in empirical software engineering. Empirical Software Engineering 13(2):211–218

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

Terdchanakul P, Hata H, Phannachitta P, Matsumoto K (2017) Bug or not? bug report classification using n-gram idf. In: Proceedings of the International Conference on Software Maintenance and Evolution, pp 534–538

Viera AJ, Garrett JM (2005) Understanding interobserver agreement: the kappa statistic. Family Medicine 37(5):360–363

Wattanakriengkrai S, Maipradit R, Hata H, Choetkiertikul M, Sunetnanta T, Matsumoto K (2018) Identifying design and requirement self-admitted technical debt using n-gram idf. In: Proceedings of the International Workshop on Empirical Software Engineering in Practice, pp 7–12

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

Yan M, Xia X, Shihab E, Lo D, Yin J, Yang X (2019) Automating change-level self-admitted technical debt determination. IEEE Transactions on Software Engineering

Zampetti F, Noiseux C, Antoniol G, Khomh F, di Penta M (2017) Recommending when design technical debt should be self-admitted. In: Proceedings of the International Conference on Software Maintenance and Evolution, 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 International Conference on Mining Software Repositories, pp 526–536

de Freitas Farias MA, de Mendonça Neto MG, da Silva AB, Spínola RO (2015) A contextualized vocabulary model for identifying technical debt on code comments. In 2015 IEEE 7th International Workshop on Managing Technical Debt (MTD), pp 25–32.

de Freitas Farias MA, Santos JA, Kalinowski M, Mendonça M, Spínola RO (2016) Investigating the Identification of Technical Debt through Code Comment Analysis. In International Conference onEnterprise Information Systems, Springer, pp 284–309.

Flisar J, Podgorelec V (2019) Identification of Self-Admitted Technical Debt Using Enhanced Feature Selection Based on Word Embedding. IEEE Access, 7, 106475–106494.

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. In ACM Transactions on Software Engineering and Methodology (TOSEM).

Liu Z, Huang Q, Xia X, Shihab E, Lo D, Li S (2018) SATD detector: a text-mining-based self-admitted technical debt detection tool. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE ’18).

Silva MC, Valente MT, Terra R (2016) Does technical debt lead to the rejection of pull requests?. In: Proceedings of the 12th Brazilian Symposium on Information Systems, ser. SBSI 16, pp 248–254.