An Empirical Study On the Removal of Self-Admitted Technical Debt

***Abstract*—Technical debt refers to the phenomena of taking shortcuts to achieve short term gain at the cost of higher maintenance efforts in the future. Recently, approaches were developed to detect technical debt through code comments, referred to as Self-Admitted Technical debt (SATD). Due to its importance, several studies have focused on the detection of SATD and examined its impact on software quality. However, preliminary findings showed that in some cases SATD may live in a project for a long time, i.e., more than 10 years. These findings clearly show that not all SATD may be regarded as 'bad' and some SATD needs to be removed, while other SATD may be fine to take on. Hence, the question becomes - which SATD should be removed?**

Therefore, in this paper, we study the removal of SATD. In an empirical study on five open source projects, we examine how much SATD is removed and who removes SATD? We also investigate for how long SATD lives in a project and what activities lead to the removal of SATD? Our findings indicate that the majority of SATD is removed and that the majority is self-removed (i.e., removed by the same person that introduced it). Moreover, we find that SATD can last between approx. 18 - 172 days, on median. Finally, through a developer survey, we find that developers mostly use SATD to track future bugs and areas of the code that need improvements. Also, developers mostly remove SATD when they are fixing bugs or adding new features. Our findings contribute to the body of empirical evidence on SATD, in particular evidence pertaining to its removal.

***Keywords*-Self-Admitted Technical Debt, Source Code Quality, Mining Software Repositories**

I. Introduction

The term technical debt was first coined by Cunningham in 1993 to refer to the phenomena of taking a shortcut to achieve short term development gain at the cost of increased maintenance effort in the future [7]. The technical debt community has studied many aspects of technical debt, including its detection [38], impact [37] and the appearance of technical debt in the form of code smells [12]. Most recently, the notion of self-admitted technical debt (SATD) has been introduced by Potdar and Shihab [26]: SATD refers to the situation when developers clearly indicate presence of technical debt in the system implementation artifacts such as source code comments. SATD refers to the situation where developers know that the current implementation is not optimal and write comments alerting the inadequacy of the solution.

Eventhough previous work argues that SATD has negative impact on software [17], [34], it has also showed that some SATD remains in a project for long periods of time (up to 10 years) after its introduction [26]. Therefore, an important question becomes "why does some SATD remain for so long?" and whether "all SATD needs to be paid back?". Examining the removal of SATD can shed light on potentially healthy patterns of debt, that may need not be paid back.

Hence, in this paper we perform a mix-match empirical study of large open source software projects, and examine phenomena relating to the removal of SATD. In particular, we examine the following questions:

RQ 1: *How much self-admitted technical debt gets removed?* Non-removal of SATD suggests relative lack of importance of SATD for the developers.

RQ 2: *Who removes self-admitted technical debt? Is it most likely to be self-removed or removed by others?* One would expect the person that introduced SATD is better aware of the presence of SATD, and, hence, a priori, is more likely to remove SATD, i.e., to pay it back.

RQ 3: *How long does self-admitted technical debt survive in a project?* Continuing the distinction between developers removing their own SATD as opposed to those removing SATD introduced by others, we would expect the former to remove SATD faster than the latter.

RQ 4: *What activities lead to the removal of self-admitted technical debt?* Developers conduct both activities such as refactoring or code improvement that might explicitly target removal of technical debt, and activities related to new functionality or bug fixing that might lead to SATD removal as a byproduct.

To answer the aforementioned questions, we leverage a NLP-based technique to determine SATD introduction and removal. In total, we examine 5,733 SATD removals in five large open source systems. Our findings indicate that 1) the majority of self-admitted technical debt comments are removed and in the studied projects the removal ranges between 40.5–90.6%, and on average 74.9% of the identified self-admitted technical debt is removed; 2) most self-admitted technical debt (on average 54.4% and median 61.0%) is self-removed; 3) the median amount of time that self-admitted technical debt stays in the project ranged between 82-613.2 days on average and 18.2-172.8 days; and 4) developer add self-admitted technical debt to track potential future bugs and code that needs improvements, whereas, developers mostly remove self-admitted technical debt when they are fixing bugs or adding new features. Very seldom do developers remove self-admitted technical debt as part of refactoring efforts or dedicated code improvement activities.

Our empirical findings provide insights to developers and software projects on how to best manage self-admitted technical debt. For example, our finding showing that most self-

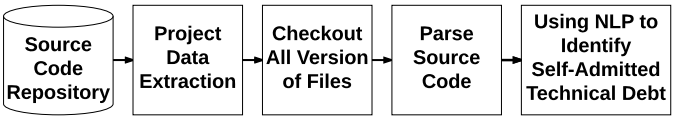

Fig. 1: Automatically Classified Data Approach Overview

admitted technical debt is self-removed provides insights on who typically removes self-admitted technical debt in large open source projects.

The rest of the paper is organized as follows: after detailing the case study setup in Section II we presents the results in Section III. We position our results with respect to the related work in Section IV and evaluate threats to validity in Section V. Finally, Section VI concludes and sketches future work.

II. Case Study Setup

The main goal of our study is to understand the removal of self-admitted technical debt. As this is, to the best of our knowledge, the first study of SATD removal, our work is exploratory in its nature. Therefore, we conduct a case study and triangulate our findings by means of a survey as recommended in the empirical software engineering literature [9], [28]. We postpone the discussion of the survey to Section III.

We checkout all versions of five large, well-commented, open source projects. Then, we use the technique recently presented by Maldonado *et al.* [22] to detect self-admitted technical debt based on the source code comments. Once self-admitted technical debt has been identified we can also conclude when and by whom has it been introduced and removed. Figure 1 shows an overview of our approach, and the following subsections detail each step.

A. Project Data

We start by selecting case study projects. While sometimes even a single case might be sufficient, *e.g.*, when it is typical, we study several systems as multiple case design is known to usually offer greater validity [9]. We select projects to cover different application domains, system sizes and numbers of contributors. Furthermore, since the self-admitted technical debt identification heavily depends on source code comments, we selected *well-commented* projects, and since we are interested in changes in self-admitted technical debt (introduction and removal) we have focused on *highly active* projects. All projects are developed in Java and use Git.

Table I provides details about each of the projects used in our study. The columns of Table I include the number of extracted comments (*i.e.,* from all versions), the number of comments analyzed after applying our filtering heuristics (*i.e.,* removing commented source code, license comments and Javadoc comments), the number of comments that were classified as self-admitted technical debt and finally the number of unique self-admitted technical debt comments. To calculate the number of unique self-admitted technical debt comments we take in consideration only the first time that the comment appears on any of the different file versions. This is necessary because the same comment may appear in different versions of the file. In total, we obtained 7,749,969 comments, found in 446,775 different versions of 30,915 Java classes. The size of the selected projects varies between 30,287 and 800,488 SLOC, and the number of contributors of these projects ranges from 32 to 289. Since there are exist different definitions for the SLOC metric we clarify that, in our study, a source line of code contains at least one valid character, which is not a blank space or a source code comment. In addition, we only use the Java files to calculate the SLOC, and to do so, we use the SLOCCount tool [35].

The number of contributors and the level of activity was extracted from OpenHub, an on-line community and public directory that offers analytics, search services and tools for open source software [2]. Number of contributors is calculated by counting the different authors that committed changes to the source code repository. However, there is the possibility that one developer possesses more than one user name in the source code repository. To mitigate this risk, OpenHub provides an interface where the manager of the project on OpenHub can link two or more different user names belonging to the same user [1].

The number of comments shown in Table I for each project does not represent the number of commented lines, but rather the number of Line, Block and Javadoc comments.

B. Checkout All Versions of Files

Since we focus on introduction and removal of self-admitted technical debt historical information about the project files is sought. First, we identify all Java source code files currently available in the latest version of the project. Then, we analyze the source code repository to track all changes done to each file. Each change made to a file produces a different version of that file, and by extracting them we can analyze each file version looking for self-admitted technical debt. Git is capable of tracking renamed or moved files based on a similarity threshold [4], [15]. In our study, we use this similarity threshold at 90%, *i.e.*, if a file is renamed or moved to another folder, and is at least 90% similar to an older version, Giet will consider that the file was just moved or renamed. However, if after the change the file is less than 90% similar with the previous version Git will consider that the original file was removed and a new one was created instead. Whitespace or blank lines are not taken into consideration to calculate the similarity of the files. Using this Git feature, we extract all versions of the Java files that are currently in the repository.

The second step to checkout all versions of files is to identify the files that are no longer present in the repository (*i.e.,* deleted files). Using Git we obtain the list of hashes of commits that has deleted at least one file and the fully qualified names of these files. Using this information we repeat the process described above to obtain all the older versions of the files. To guarantee the correctness of the process we focus solely on Java source files, and ensure that every fully qualified path of the file is analyzed only once.

TABLE I: Details of Studied Projects

Project	Project details				Comments details			
	#Java files	SLOC	#file versions	#contributors	#comments	#comments after filtering	#TD comments	#unique TD comments
Camel	15,091	800,488	254,920	289	1,634,361	700,412	20,141	4,331
Gerrit	3,059	222,476	53,298	270	1,018,006	129,023	4,810	271
Hadoop	8,466	996,877	79,232	160	2,512,673	1,172,051	18,927	1,164
Log4j	1,112	30,287	12,609	35	248,276	61,690	1,893	135
Tomcat	3,187	297,828	46,716	32	2,336,653	1,081,492	26,725	1,317

Once this step is complete, we have at our disposal the information regarding the files and their versions stored in the database, and an actual copy of each file version in a structured directory inside our tool that will be used to extract the remainder of the data for our study.

C. Identifying self-admitted technical debt comments

We parse the source code in order to extract the comments. We use an open source library SrcML [6] to parse the source code, and extract the comments and the information related to them such as the line that each comment starts, finishes and the type of the comment (*i.e.,* Javadoc, Line or Block).

Next, we use the technique presented in by Maldonado *et al.* [22] to identify the self-admitted technical debt comments. We refer readers to Maldonado *et al.*'s paper for full details on how to identify self-admitted technical debt comments, however, to make our paper self-contained, we highlight the key points of their approach next.

1) Filtering Irrelevant Comments.: As the prior work showed, not all comments can contain self-admitted technical debt [26], [34]. Therefore, as the pre-processing we remove the following types of comments from our dataset:

- **License comments** that generally do not contain self-admitted technical debt and are commonly located before the class declaration. That said, comments that contain task annotations (*i.e.,* "TODO:", "FIXME:", or "XXX:") [33] are not removed since they are usually leveraged by most IDEs, e.g., Eclipse and Netbeans, to automatically generate task lists.
- **Commented source code** is also ignored since prior work showed that it generally does not contain self-admitted technical debt [26], [34].
- **Automatically generated comments** by the IDE are also removed since they, by definition, do not indicate self-admitted technical debt.
- **Javadoc comments** are also removed since they rarely mention self-admitted technical debt [26].

The pre-processing steps above significantly reduce the number of comments in the dataset and allow us to focus on the most applicable and insightful comments. For example, as shown in Table I, in the Camel project, applying the above steps helped to reduce the number of comments from 1,634,361 to 700,412 meaning a reduction of 57.1% in the number of comments to be classified. Using the filtering heuristics we were able to eliminate between 53.3% to 87.3% of all comments. Table I provides the number of comments kept after the filtering heuristics for each project.

2) Applying the NLP Classifier.: We use the Stanford NLP Classifier [24] to classify self-admitted technical debt comments. The NLP Classifier takes as input classified data items (comments), and automatically learns *features* (*i.e.,* words) from each *datum* that are associated with positive or negative numeric *votes* for each class. The weights of the features are learned automatically based on the manually classified training data items (supervised learning). The Stanford NLP Classifier builds a *maximum entropy model* [25], which is equivalent to a multi-class regression model, and is trained to maximize the conditional likelihood of the classes taking into account feature dependences when calculating the feature weights.

To train the NLP classifier, we used the manually classified self-admitted technical debt comments data set provided by Maldonado *et al.* [22]. The data set contains contains 62,566 comments extracted from ten open source projects. These comments were classified as self-admitted technical debt comments or as regular comments (*i.e.,* comments without self-admitted technical debt). The manually classified data set was verified by the authors of the paper and they showed that two independent reviewers agreed on the classification, achieving a Cohen's Kappa value of +0.81. In addition, the work by Maldonado *et al.* [22] showed that the NLP classifier is correct in identifying self-admitted technical debt with an average precision of 0.72 and recall of 0.56. Although these precision and recall values may not seem high, they do represent the state of the art and outperform the comment-patterns technique, which all prior work was built on top of (i.e., [3], [26], [34]) by 230%, on average.

III. Case Study Results

The main goal of our study is to better understand what happens to self-admitted technical debt once it is introduced into software projects. To do so, our first step is to quantify how much of self-admitted technical debt comments gets removed (RQ1). Next, we analyze who removes self-admitted technical debt, i.e., if the same developer that introduced the debt is also most likely to remove it (RQ2). Then, we investigate how long the self-admitted technical debt remains in the project (RQ3). Finally, we conduct a survey with 14 developers to understand why self-admitted technical debt is

introduced and removed (RQ4). For each question, we describe the motivation behind the question, the approach to address the question, and present the results.

RQ1. How much self-admitted technical debt gets removed?

Motivation: Previous work showed that technical debt is widespread, unavoidable, and has a negative impact on the quality of software projects [21]. Therefore, a priori we expect that removing technical debt is a concern for developers. To understand how developers deal with technical debt we must first quantify how much debt is removed.

Approach: To answer this question we automatically identify self-admitted technical debt from the five analyzed projects. As described in Section II-B, we stored all versions of all source code files. Then, for each analyzed self-admitted technical debt comment we take the oldest file version available in which the debt was found and incrementally search for matches in future versions of the file. The first time that the analyzed self-admitted technical debt comment(s) appears in a file, indicates the exact file version that the self-admitted technical debt comment was introduced. To analyze if the introduced self-admitted technical debt comment was later removed, we search for the comment in the remaining file versions. When the comment is no longer found, we mark that version of the file as the removal version. In certain cases, a self-admitted technical debt comment is found in one version only (i.e., the version that it is introduced in). Such cases indicate a scenario where the self-admitted technical debt was introduced and removed immediately after.

Results: Table II presents the identified and removed self-admitted technical debt comments. We find that the majority (*i.e.,* on average 74.4%, median 76.7%) of the identified self-admitted technical debt comments were removed. We measure the average on a per project basis, *i.e.,* the total from each project is taken and the average over the five projects in provided. For example, we were able to find 271 unique instances of self-admitted technical debt comments when analyzing the Gerrit project. 76.7% (*i.e.,* 208) of these self-admitted technical debt comments were removed during the evolution of the project. Camel had the highest self-admitted technical debt comments removal percentage (*i.e.,* 90.6), whereas Hadoop had the lowest removal percentage achieving 40.5%.

Our findings indicate that developers tend to be aware and do care about self-admitted technical debt. This finding corroborates with the survey findings of Ernst *et al.* [10].

The majority of self-admitted technical debt comments are removed over time. In our five case study projects, between 40.5–90.6% (median 76.7%) of the identified self-admitted technical debt is removed.

RQ2. Who removes self-admitted technical debt? Is it most likely to be self-removed or removed by others?

Motivation: As opposed to the technical debt in general, *self-admitted* technical debt stands for technical debt "confessed" by the developers themselves. This intuition leads us to believe that it would be natural that the developers who expressed concern about the code would be also the ones who fix it in the future. However, it is unknown whether this is the case. It makes intuitive sense that self-removal of self-admitted technical debt is easier, since the developers know about the reason for the self-admitted technical debt introduction and possibly how to address it. The findings of this question have implications on the way that developers/manager/projects need to manage self-admitted technical debt. For example, if it is found that self-admitted technical debt is mostly addressed by others, then projects need to pay special attend to how this technical debt (and the areas of the code that it exists in) is documented. If on the other hand, it is indeed mostly self-removed, then the problem is less troubling.

TABLE II: Removed Self-Admitted Technical Debt per Project

Project	#identified	#removed	% removed	% remains
Camel	4,331	3,926	90.6	9.4
Gerrit	271	208	76.7	23.3
Hadoop	1,164	472	40.5	59.5
Log4j	135	118	87.4	12.6
Tomcat	1,317	1,009	76.6	23.4
Average	-	-	74.4	25.6
Median	-	-	76.7	23.3

TABLE III: Self-Removed Technical Debt per Project

Project	#removed	#self-removed	% self-removed
Camel	3,926	2,652	67.5
Gerrit	208	149	71.6
Hadoop	472	116	24.6
Log4j	118	72	61.0
Tomcat	1,009	578	57.3
Average	-	-	54.4
Median	-	-	61.0

Approach: To answer this question we analyzed the authors of the changes (*i.e.,* commits from the source code repository) that introduced or removed self-admitted technical debt comments. In order to do that, we first determine the commit in which a self-admitted technical debt comment was added, then we check the further file versions to determine if there is any commit that removed the self-admitted technical debt comment. Finally, we compare the authors of the commits to see if they are the same or not.

We take into consideration two attributes of the change when comparing authors—the author name and email address. This is a necessary heuristic to mitigate the risk of misclassifying authors that change their names in the source code repository during the evolution of the project.

Results: Table III shows that in most cases, the majority of self-admitted technical debt is removed by the same author who introduced it, referred to as "self-removed technical debt". On average, 54.4% of all removals are self-removed and in

four of the five projects, self-removal accounts for more than 50%. Once again, we measure the average on a per project basis, *i.e.*, the total from each project is taken and the average over the five projects in provided. The project with highest percentage of self-removed technical debt was Gerrit with 71.6%, with the lowest percentage—Hadoop with 24.6%.

Hadoop tends to be an outlier in terms of self-removed self-admitted technical debt, however, it is worth mentioning that Hadoop had the least amount of removals overall (only 40.5% of the self-admitted technical debt is ever removed). There are many possible reasons for the low removal rates, *e.g.*, high developer churn or lack of process to deal with technical debt. Although we shed some light on the potential reasons for the removal of self-admitted technical debt later in RQ4, we believe that determining the exact reasons for self-admitted technical debt removal warrant a study on its own.

> ***The majority of self-admitted technical debt is self-removed. On average 54.4% of self-admitted technical debt is self-removed and on median 61.0% is self-removed.***

RQ3. How long does self-admitted technical debt survive in a project?

Motivation: We know that the majority of self-admitted technical debt is removed and most of the time it is removed by the same developer who introduced it. Next, we would like to know how long self-admitted technical debt lives in a project before it is actually removed. Answering this question helps us to understand for how long it is normal to have self-admitted technical debt comments in the projects. In addition, once we quantify the number of self-removed technical debt and the number of non self-removed technical debt comments, we would like to understand if these two categories of removal have differences between them. For example, since we know that the majority of self-admitted technical debt is self-removed, is it the case that it is removed faster since our intuition is that it would be easier to address.

Approach: To determine the amount of time that self-admitted technical debt lives in a project, we use the time difference between the commit that introduces and removes the self-admitted technical debt comment. The steps to identify the self-admitted technical debt introducing and removing commits are the same as we outlined in RQs 1 and 2. We measure the average and median time for self-admitted technical debt to be removed.

Additionally, we generate survival plots for the removal of self-admitted technical debt to determine how likely the technical debt will live in a project. Survival plots show the (general) trend times for a given even to occur. In our case, the survival plots show the percentage of self-admitted technical debt that survives in a project overtime. Finally, we distinguish between self- and non-self-removed technical debt and compare the removal time of each. We compare the two distributions (*i.e.*, self- and non-self-removal) using a Mann-Whitney test [23] to determine if the difference is statistically significant at the customary level of 0.05.

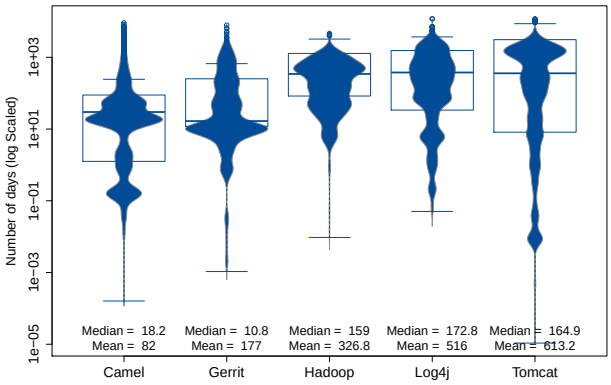

Fig. 2: The distribution of all the removed STD comments

We also estimated the magnitude of the difference between self-removed technical debt and non-self-removed technical debt using the Cliff's Delta (or d) [14], a non-parametric effect size measure for ordinal data. We consider the effect size values: small for $d < 0.33$ (positive as well as negative values), medium for $0.33 \leq d < 0.474$ and large for $d \geq 0.474$.

Results: Figure 2 shows the mean and median times for self-admitted technical debt to be removed from the respective projects. The distribution of self-admitted technical debt removal is skewed, as indicated by plots and the difference in the mean and median removal times. In general, the time that self-admitted technical debt stays in a project varies from one project to another and varies between 18.2–172.8 days on median and 82–613.2 days on average. One clear finding however, is that in Camel and Gerrit, self-admitted technical debt is removed faster than in Hadoop, Log4j and Tomcat.

Figure 3 shows the survival plots of self-admitted technical debt for the five studied projects. Survival plot is a technique originating from the medical domain indicating the probability of a patient to survive at least for x days. To estimate this probability one would ideally like to have complete information about the death time of all patients. Such an assumption is, however, usually not realistic as some patients might still be alive at the end of the observations, *i.e.*, the data is right-censored. Kaplan and Meier [18] have proposed a technique to estimate the survival in presence of right-censored data. As we have seen in Table II some self-admitted technical debt comments remained at the end of the observations, *i.e.*, our data is also right-censored, in Figure 3 we present the Kaplan-Meier estimators. The use of Kaplan-Meier estimators is common in software evolution applications of survival analysis [13], [29].

Inspecting Figure 3 we observe that for all projects, there is a steep decline in the first few hundred days, suggesting that in all projects an important share of self-admitted technical debt is rapidly removed. Projects do differ in how steep the drop is and where it flattens out. For example, for the Camel project, there is a steep drop in self-admitted technical debt

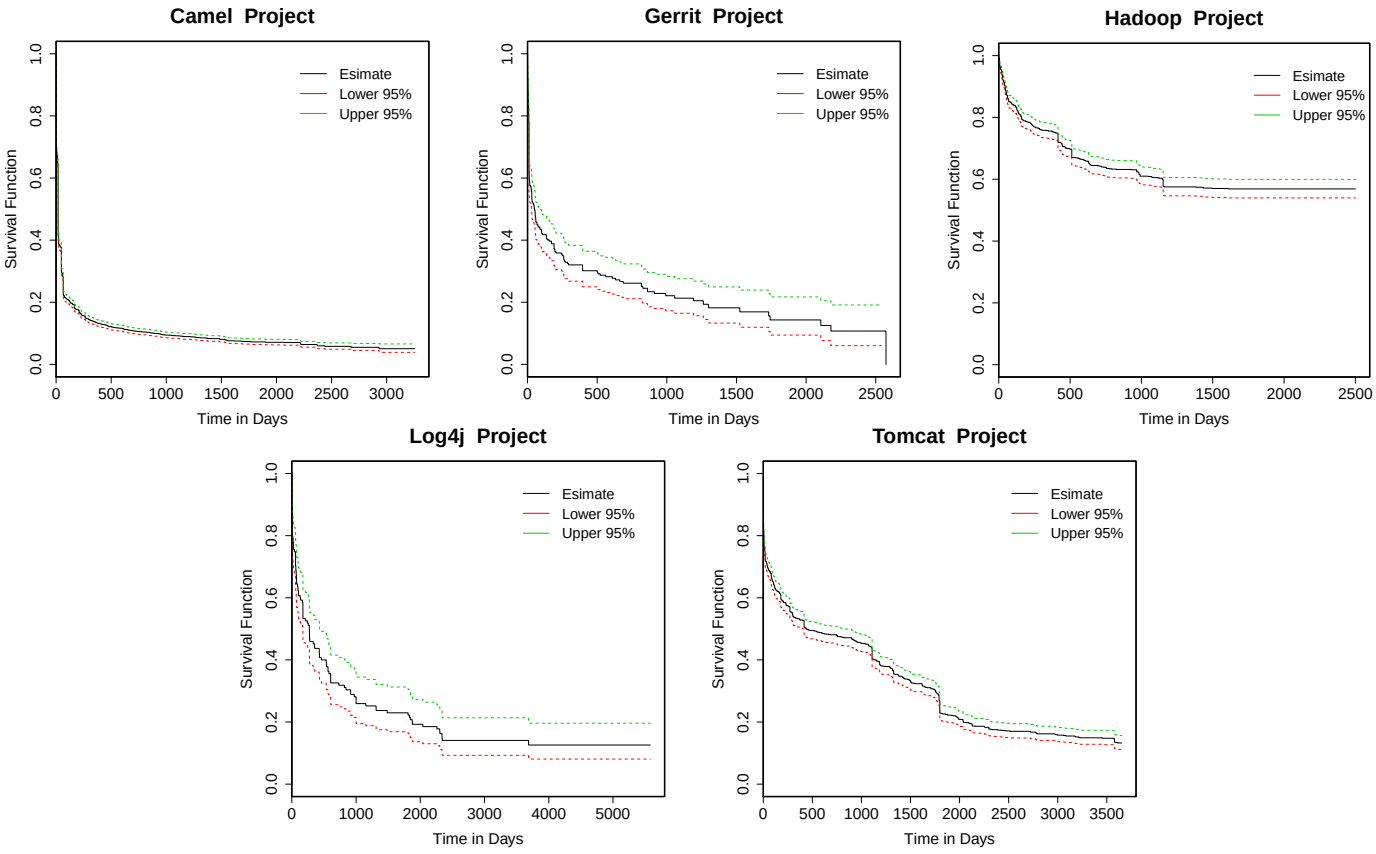

Fig. 3: Survival plots show the probability of the removal of STD Comment for all studied projects

after around 126 days and a long tail after that. This means that in Camel, the likelihood of self-admitted technical debt surviving (*i.e.*, existing in the system after introduction) drops significantly after 126 days, and after that time, the chance of surviving is less than 20%, as indicated by the survival function. Another extreme case is Hadoop, where the chance of self-admitted technical debt surviving for more than 2,000 days is very high, close 59.5%, the percentage of the self-admitted technical debt comments remaining at the end of the observations reported in Table II.

We also compare the time that self-removed and non self-removed self-admitted technical debt exists in the system before it gets removed. We find that self-removed technical debt gets removed faster than non self-removed technical debt. Figure 4 shows that, on median for all projects, self-removed technical debt is removed earlier than non-self-removed technical debt. Our finding confirms our intuition, however, the exact reasons (e.g., is it because the remover is more familiar with the debt) as to why self-removals take less time warrant a study on its own. Table IV shows the result of Mann-Whitney test and Cliff's Delta (d), which is a measure of effect size. We observe that for all the studied projects the difference between self-removed and non-self-removed technical debt is statistically different. Also, the effect size, for all the projects is large except for Camel, in which case the effect size is medium.

TABLE IV: Self-Removal vs Non Self-Removal: Mann-Whitney Test (p-value) and Cliff's Delta (d)

Project	p**-value**	d
Camel	0.0001253	−0.075(medium)
Gerrit	3.581e-14	−0.671(large)
Hadoop	2.2e-16	−0.531(large)
Log4j	2.345e-06	−0.517(large)
Tomcat	2.2e-16	−0.820(large)

The amount of time self-admitted technical debt remains in a project before removal varies from one project to another and ranges between 18.2–172.8 days on median and 82–613.2 days on average. Moreover, self-removed technical debt is removed faster than non-self-removed technical debt.

RQ4. What activities lead to the removal of self-admitted technical debt?

Motivation: Thus far, our analysis has been quantitative in nature. To triangulate our findings and better understand our findings, we perform complementary qualitative analysis to understand the experiences and motives of developers who introduce and remove self-admitted technical debt.

Fig. 4: Self-Removal vs Non Self-Removal for all studied projects

Approach: To understand the activities that lead to the introduction and removal of self-admitted technical debt, we designed an online survey. The survey included three main sections: 1) questions regarding the participant's role and development tasks and experience in the project, 2) three Likert-scale questions about the frequency of developers encountering, adding, and addressing self-admitted technical debt, and 3) two open ended questions asking why developers add or remove self-admitted technical debt.

To identify the target population, we collected the names and emails of all developers who added or removed self-admitted technical debt in the five studied projects and an additional two projects from our training dataset, namely Apach-Ant and Jmeter. We chose these two additional projects to increase the potential number of respondents and did not want to include all of the training dataset projects to not overwhelm developers with requests for surveys.

In total, we found 250 unique developers from the studied projects and we successfully sent the survey to 188 of them. We received 14 responses, *i.e.*, the response rate is 7.4%. Although this is lower than the response rate reported in software engineering surveys [27], the area of technical debt is difficult to discuss, especially since some developers may feel they will be negatively perceived. For, the open-ended questions, we manually analyzed the free-text answers and identified six main reasons why developers add self-admitted technical debt and five main reasons for removing self-admitted technical debt.

Table V shows the respondent's role in the projects, their main development tasks and experience. Of the 14 respondents, eight identified themselves as core developers, and six as contributors to the projects. Five of the 14 respondents work on fixing bug, and five work on implementing new features. Only one respondent has the task of code reviewer. Another three respondents indicted having different tasks (*e.g.*, project user). Twelve respondents have more than five years of experience.

Results: Figure 5 shows the results of the Likert-scale questions about how often developers encounter, add, address

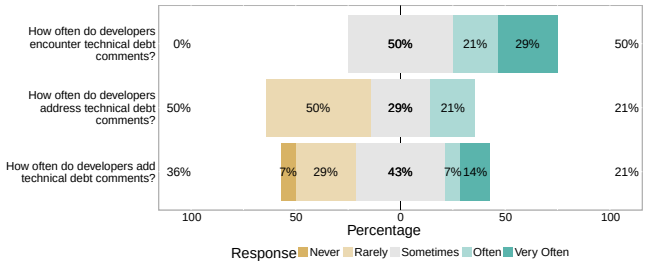

Fig. 5: Survey responses on how often do developers encounter, add and address self-admitted technical debt.

TABLE V: Background of participants in online survey

Role	Developer Tasks	Developer Exp. (in years)		
		1-2	3-5	>5
Contributor (8)	BF, BF, NF, OTH, BF, BF, OTH, NF	1	0	7
Core Developer (6)	NF, BF, OTH, NF, CR, NF	0	1	5

BF=Bug Fixing, NF=New Feature, CR=Code Review, OTH=Other.

self-admitted technical debt. Developers mostly agreed that they encounter source code comments indicating self-admitted technical debt. All respondents report that they encounter self-admitted technical debt comments at least as often as add them or address them. Interestingly enough six respondents indicate that they add self-admitted technical debt comments more often than address them, while only three indicate that they address self-admitted technical debt comments more often then add them.

As for why developers tend to add self-admitted technical debt, nine respondents (P1, P4, P5, P8, P9, P11, P12, P13, and P14) indicted that they add self-admitted technical debt as *a tracker in the source code for potential bugs or source code that needs to be improved or document a need for a new feature*. For example, P12 states that *"It is usually a marker in the source of a missing feature or known bug."*. Also, contrary to Postdar and Shihab [26] we find that developers add self-

admitted technical debt because of time pressure (P1, P2, P7, P13, and P14) to deliver tasks. For example, P1 *"Because they want to deliver, and when balancing an early delivery against technical debt."* Some other reasons for adding self-admitted technical debt are very rare and are only mentioned once or twice (e.g,. a remainder or looking for feedback). For example, P5 said that *"They are not sure about the effects of their code and want feedback..."*.

In response to the question on why developers address self-admitted technical debt, we identified five reasons. The most cited reason for addressing self-admitted technical debt is *to fix bugs* (P1, P4, P5, P7, P8, P9, P10, P12 and P13). For example, P12 states *", usually as part of fixing a user-reported issue..."* The second most frequent reasons is to add a new feature (P1, P4, P6, P12, and P14) and improve the code overall (P7, P8, P9, P10, and P11). The other two, less frequent, reasons are addressing self-admitted technical debt when refactoring code (P7 and P9) and to provide a generally better solution (P2 and P7). Our findings indicate that there is a need for software projects to allocate resources to specifically address self-admitted technical debt, since most respondents do not seem to indicate that a systematic process is in place to address self-admitted technical debt. And, in most cases it seems like dealing with self-admitted technical debt is done in an ad-hoc manner.

Developer add self-admitted technical debt to track potential future bugs, code that needs improvements or areas to implement new features. Developers mostly remove self-admitted technical debt when they are fixing bugs or adding new features. Very seldom do developers remove self-admitted technical debt as part of refactoring efforts or dedicated code improvement activities.

IV. Related Work

In this section, we describe the related work. We divided the related work into two sections; work related to the management and the detection of technical debt in general and work related to the identification of self-admitted technical debt.

The detection & management of technical debt in general. A number of earlier studies studied the management and detection of technical debt in general. Seaman *et al.* [30], Kruchten *et al.* [20] and Brown *et al.* [5] made several reflections about the term 'technical debt' and mentioned that it is commonly used to communicate development issues to managers. Other work by Zazworka *et al.* [38] focused on the detection of technical debt, where they conducted experiments to compare the efficiency of automated tools in comparison with human elicitation in detecting technical debt. They found that there is a small overlap between the two approaches. They also concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt. In follow up work, Zazworka *et al.* [37] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of debt, namely design debt measured using God classes. They found that God classes are more likely to change, and therefore, have a higher impact on software quality. Other work by Fontana *et al.* [12] investigated design technical debt appearing in the form of code smells, namely God Classes, Data Classes and Duplicated Code. They proposed an approach to classify which one of the different code smells should be addressed first, based on thier potential risk. Ernst *et al.* [10] conducted a survey involving more than 1,800 participants and found that architectural decisions are the most important source of technical debt.

In an earlier study Klinger *et al.* have conducted four interviews and observed that "the individuals choosing to incur technical debt are usually different from those responsible for servicing the debt" [19]. This observation has been questioned by Spinola *et al.*, who have found that while the online-survey respondents tended to agree with the observation, the paper-survey respondents achieved high consensus in neither agreeing nor disagreeing with the observation [32]. Therefore, to complement these studies we analyze the source code. Indeed, if the observation of Klinger *et al.* holds for self-admitted technical debt then we expect to see a clear separation between the individuals introducing self-admitted technical debt and individuals removing self-admitted technical debt, resulting in a low self-admitted technical debt self-removal.

Jian and Hassan have studied removal of comments in PostgreSQL from 1996 to 2005 [16]. They have observed that at each 30-day period 0–40% of functions with header comments have been removed; similar observation has been made for functions with non-header comments. Unfortunately, different focus of our studies (comments vs. functions, self-admitted technical debt-comments vs. any comments) render our results incomparable.

Our work is differs from the work that uses code smells to detect design technical debt, since we use code comments to detect technical debt. Moreover, our study focuses on the removal of self-admitted technical debt, rather than its identification or management.

The detection & management of "self-admitted" technical Debt. Potdar and Shihab [26] introduce the self-admitted technical debt. They extracted code comments of four projects and analyzed more than 100K comments to come up with 62 patterns that indicate self-admitted technical debt. Their findings show that 2.4% - 31% of the files in a project contain self-admitted technical debt. In a recent work, Maldonado *et al.* [22] use natural language processing technical to identify self-admitted technical debt from source code comment. Their experiment showed that the proposed method achieved 90% classification accuracy in identifying design and requirement self-admitted technical debt. Bavota and Russo [3] replicated the study of self-admitted technical debt on a large set of Apache projects and confirmed the findings observed by Potdar and Shihab in their earlier work [26]. Maldonado and Shihab [8] examined more than 33K comments to classify the different types of self-admitted technical debt found in source code comments. Other work Farias *et al.* [11] proposed a

contextualized vocabulary model for identifying technical debt in comments using word classes and code tags in the process.

Other work studied the management and the impact of self-admitted technical debt. Wehaibi *et al.* [34] examined the impact of self-admitted technical debt and found that self-admitted technical debt leads to more complex changes in the future. All three of the aforementioned studies used the comment patterns approach to detect self-admitted technical debt. Kamei *et al.* [17] proposed a method to measure technical debt interest using self-admitted technical debt comments in the source code. They found around 42% of the technical debt in the studied projects incurs positive interest.

Similar to the prior work, our work also uses code comments to detect self-admitted technical debt. However, we use the NLP technique to identify self-admitted technical debt in order to conduct our empirical study on the *removal* of self-admitted technical debt.

V. Threats to Validity

Following common guidelines for empirical studies [36], this section discusses the threats to validity of our study.

Internal Validity: Internal validity concerns with factors that could have influenced our results. We rely on the NLP classification to determine self-admitted technical debt. As mentioned earlier, this approach is not perfect, achieving an average precision of 0.72 and recall of 0.56. Although the precision and recall values are not very high, the NLP technique is considered the state-of-the-art in detecting self-admitted technical debt. The NLP technique outperforms the comment-patterns technique, which all prior work was built on top of (i.e., [3], [26], [34]) by 230%, on average. We train the Stanford NLP classifier on manually tagged self-admitted technical debt comments provided in prior work. The manually classified comments have been verified and published in peer-reviewed venues.

To understand the type of activities that lead to the introduce and remove of self-admitted technical debt, we conducted a online survey. We sent the survey to 188 developers who responsible for adding and removing self-admitted technical debt, and we received 14 (7.4%) responses which may be considered small. However, a 7.4% response rate is considered to be an acceptable response rate in questionnaire-based software engineering surveys [31].

Construct Validity: Threats to constructed validity concern the relationship between theory and observation. To identify the removed and added self-admitted technical debt, we consider commits as a single unit of change. However, a single commit may contains other source code changes. Moreover, we rely on Open-hub's data to merge developer identities, hence, our study is only as accurate as Open-hub's classification.

External Validity: Threats to external validity concern the generalization of our findings. Our study is conducted on five large open source projects and contains more than 5,700 comment removals. That said, our findings may not generalize to other open source or commercial systems.

VI. Conclusion and Future work

Self-admitted technical debt refers to technical debt that can be detected through code comments. Prior work examined the detection, mangement and impact of self-admitted technical debt. However, little is known about the removal of such technical debt. In this paper, we conduct and empirical study to examine how much self-admitted technical debt is removed, how long such technical debt lives in a project before removal and who removes such debt. We find that the majority of self-admitted technical debt is removed (74.4% on average), that self-admitted technical debt is mostly self-removed (54.4% on average), and that it lasts between 82 - 613.2 days, on average, in a project before it is removed. Then, we conduct a survey with 12 developers to understand the reasons for the introduction and removal of self-admitted technical debt. We find that there is no formal process to remove self-admitted technical debt, and most removals occur as part of bug fixing.

Our results provide insights that indicate that self-admitted technical debt is important, which is why the majority of it is is removed. Also, our results suggest that although developers are aware of the need to remove self-admitted technical debt, most project do not employ any formal process to address it, hence there is a need for techniques and studies to allow projects to effectively and systematically address self-admitted technical debt.

In the future, we plan to perform qualitative studies that examine the 'whys' of our findings. In particular, we would like to examine why developers tend to self-remove technical debt. Additionally, we plan to better understand why some projects remove less self-admitted technical debt than others.

References

[1] OpenHub Aliases Level. http://blog.openhub.net/faq-2/. Accessed: 2016-05-14.

[2] OpenHub homepage. https://www.openhub.net/. Accessed: 2016-05-14.

[3] G. Bavota and B. Russo. A large-scale empirical study on self-admitted technical debt. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR'16, pages 315–326. ACM, 2016.

[4] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. German, and P. Devanbu. The promises and perils of mining git. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*, MSR '09, pages 1–10. IEEE Computer Society, 2009.

[5] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER'10, pages 47–52. ACM, 2010.

[6] M. L. Collard, M. J. Decker, and J. I. Maletic. srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration. In *Proceedings of the 29th International Conference on Software Maintenance*, ICSM'13, pages 516–519. IEEE Computer Society, 2013.

[7] W. Cunningham. The WyCash portfolio management system. In *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA, pages 29–30. ACM, 1992.

[8] E. d. S. Maldonado and E. Shihab. Detecting and quantifying different types of self-admitted technical debt. In *Proceedings of the 7th International Workshop on Managing Technical Debt*, MTD'15, pages 9–15. IEEE, 2015.

[9] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian. *Selecting Empirical Methods for Software Engineering Research*, pages 285–311. Springer London, London, 2008.

[10] 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 *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 50–60. ACM, Sept 2015.

[11] M. A. d. F. Farias, M. G. d. M. Neto, A. B. d. Silva, and R. O. Spinola. A contextualized vocabulary model for identifying technical debt on code comments. In *Proceedings of the 7th International Workshop on Managing Technical Debt*, MTD'15, pages 25–32. IEEE, 2015.

[12] F. Fontana, V. Ferme, and S. Spinelli. Investigating the impact of code smells debt on quality code evaluation. In *Proceedings of the 3rd International Workshop on Managing Technical Debt*, MTD'12, pages 15–22. IEEE, June 2012.

[13] M. Goeminne and T. Mens. Towards a survival analysis of database framework usage in Java projects. In *ICSME*, pages 551–555. IEEE, 2015.

[14] R. J. Grissom and J. J. Kim. *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[15] H. Hata, O. Mizuno, and T. Kikuno. Historage: Fine-grained version control system for java. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, IWPSE-EVOL'11, pages 96–100. ACM, 2011.

[16] Z. M. Jiang and A. E. Hassan. Examining the evolution of code comments in postgresql. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR'06, pages 179–180. ACM, 2006.

[17] Y. Kamei, E. Maldonado, E. Shihab, and N. Ubayashi. Using analytics to quantify the interest of self-admitted technical debt. In *Proceedings of the First International Workshop on Technical Debt Analytics*, 2016.

[18] E. L. Kaplan and P. Meier. Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 53(282):457–481, 1958.

[19] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams. An enterprise perspective on technical debt. In *Proceedings of the 2nd Workshop on Managing Technical Debt*, MTD'11, pages 35–38. ACM, 2011.

[20] 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. *SIGSOFT Softw. Eng. Notes*, pages 51–54, 2013.

[21] E. Lim, N. Taksande, and C. Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Software*, 29:22–27, 2012.

[22] E. Maldonado, E. Shihab, and N. Tsantalis. Using natural language processing to automatically detect self-admitted technical debt. *IEEE Transactions on Software Engineering (TSE)*, page To Appear, 2017.

[23] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[24] C. D. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. J. Bethard, and D. McClosky. The Stanford CoreNLP natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60. Association for Computational Linguistics, June 2014.

[25] K. Nigam, J. Lafferty, and A. McCallum. Using maximum entropy for text classification. In *IJCAI-99 workshop on machine learning for information filtering*, volume 1, pages 61–67, 1999.

[26] A. Potdar and E. Shihab. An exploratory study on self-admitted technical debt. In *Proceedings of the 30th International Conference on Software Maintenance and Evolution*, pages 91–100. IEEE Computer Society, Sept 2014.

[27] T. Punter, M. Ciolkowski, B. Freimut, and I. John. Conducting online surveys in software engineering. In *Proceedings 2003 International Symposium on Empirical Software Engineering.*, ISESE '03, pages 80–88. IEEE, 2003.

[28] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering (EMSE)*, 14(2):131–164, 2009.

[29] I. Samoladas, L. Angelis, and I. Stamelos. Survival analysis on the duration of Open Source projects. *Information and Software Technology*, 52(9):902–922, 2010.

[30] C. Seaman and Y. Guo. Measuring and monitoring technical debt. volume 82 of *Advances in Computers*, chapter 2, pages 25–46. Elsevier, 2011.

[31] J. Singer, S. E. Sim, and T. C. Lethbridge. Software engineering data collection for field studies. In *Guide to Advanced Empirical Software Engineering*, pages 9–34. Springer London, 2008.

[32] R. O. Spnola, A. Vetró, N. Zazworka, C. Seaman, and F. Shull. Investigating technical debt folklore: Shedding some light on technical debt opinion. In *Proceedings of the 4th International Workshop on Managing Technical Debt*, MTD'13, pages 1–7. IEEE Press, May 2013.

[33] M. Storey, J. Ryall, R. Bull, D. Myers, and J. Singer. Todo or to bug. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE'08, pages 251–260. IEEE, May 2008.

[34] S. Wehaibi, E. Shihab, and L. Guerrouj. Examining the impact of self-admitted technical debt on software quality. In *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering*, SANER'16, pages 179–188. IEEE, 2016.

[35] D. A. Wheeler. *SLOC count user's guide*, 2004.

[36] R. K. Yin. *Case study research: Design and methods*. Sage publications, 2013.

[37] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2nd International Workshop on Managing Technical Debt*, MTD'11, pages 17–23. ACM, May 2011.

[38] N. Zazworka, R. O. Spínola, A. Vetro', F. Shull, and C. Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, EASE'13, pages 42–47. ACM, 2013.