An Exploratory Study on Self-Admitted Technical Debt

Aniket Potdar
Department of Software Engineering
Rochester Institute of Technology
Rochester, NY, USA
Email: asp6719@rit.edu

Emad Shihab
Department of Computer Science and Software Engineering
Concordia University
Montreal, QC, Canada
Email: eshihab@cse.concordia.ca

***Abstract*—Throughout a software development life cycle, developers knowingly commit code that is either incomplete, requires rework, produces errors, or is a temporary workaround. Such incomplete or temporary workarounds are commonly referred to as 'technical debt'. Our experience indicates that self-admitted technical debt is common in software projects and may negatively impact software maintenance, however, to date very little is known about them.**

Therefore, in this paper, we use source-code comments in four large open source software projects - Eclipse, Chromium OS, Apache HTTP Server, and ArgoUML to identify self-admitted technical debt. Using the identified technical debt, we study 1) the amount of self-admitted technical debt found in these projects, 2) why this self-admitted technical debt was introduced into the software projects and 3) how likely is the self-admitted technical debt to be removed after their introduction. We find that the amount of self-admitted technical debt exists in 2.4% - 31% of the files. Furthermore, we find that developers with higher experience tend to introduce most of the self-admitted technical debt and that time pressures and complexity of the code do not correlate with the amount of self-admitted technical debt. Lastly, although self-admitted technical debt is meant to be addressed or removed in the future, only between 26.3% - 63.5% of self-admitted technical debt gets removed from projects after introduction.

I. Introduction

Delivering high quality, defect-free software is the goal of all software projects. To ensure the delivery of high quality software, software project often plan their development and maintenance efforts. However, in many cases, developers are rushed into completing tasks for various reasons. A few of these reasons mentioned in prior work include, cost reduction, satisfying customers and market pressure from competition [1]. Intuition and general belief indicate that such rushed development tasks (also known as technical debt) negatively impact software maintenance and overall quality [2].

A plethora of prior work proposed techniques to support software maintenance and ensure high software quality. For example, prior work focused on understanding and predicting software defects (e.g. [3]), analyzing bug fix patterns (e.g. [4]), and attempting to understand and eliminate rework and maintenance (e.g., [5]). The majority of the aforementioned prior work used historical development data and source-code metrics to perform their studies. More recently, researchers leveraged natural language to help identify potentially problematic areas of the software. For example, work by Tan *et al.* [6] developed natural language processing tools to find comment-bug inconsistencies. Other work identified the coevolutionary relationship between source code and its associated comments (e.g., [7], [8]) and used task annotations to manage productivity [9].

The majority of the prior work focused on quality issues that are due to *unintentional* errors by developers (i.e., errors introduced by the developers are assumed to mistakes). However, to the best of our knowledge, very few prior studies examined the impact of errors that might be introduced due to *intentional* (i.e., self admitted) quick or temporary fixes (i.e., technical debt). Studying this self-admitted technical debt is important since they appear frequently in some projects (as we show later in this study) and prior work indicated that they negatively impact quality [2].

Therefore, in this paper we perform an exploratory study to better understand self-admitted technical debt. Inspired by prior work (e.g., [6], [8], [10], [11]), we use source-code comments to detect self-admitted technical debt. We perform our study on four large open source projects - namely Eclipse, Chromium OS, ArgoUML and Apache httpd. We focus on quantifying the amount of self-admitted technical debt (RQ1), on determining why self-admitted technical debt is introduced (RQ2) and how much of self-admitted technical debt is actually removed after their introduction (RQ3).

We make the following contributions:

- **Identify comment patterns that indicate self-admitted technical debt.** We manually read through 101,762 code comments to determine patterns that indicated self-admitted technical debt. In the end, we identified 62 different comment patterns that indicate self-admitted technical debt.
- **Measure how much self-admitted technical debt exists, why self-admitted technical debt is introduced and how much self-admitted technical debt is removed after their introduction.** We find that 2.4% - 31.0% of the files contain self-admitted technical debt, that more experienced developers introduce more self-admitted technical debt and that self-admitted technical debt is introduced throughout their development activity (i.e., they do not only introduce self-admitted technical

debt during the beginning or end of their development activity). Also, we find that time to release and complexity of the code are not strongly associated with self-admitted technical debt and that even after many releases only between 26.3% - 63.5% of self-admitted technical debt is removed.

- **Contribute a rich data set of self-admitted technical debt.** In order to encourage future research in the area of self-admitted technical debt, we make our dataset from this study publicly available[1].

The rest of the paper is organized as follows. Sections II presents the related work. We setup our case study in Section III and describe our approach. Section IV presents our case-study results. In section V we analyze the instances when code and comments evolve consistently and instances when either code or comments are updated inconsistently. Section VI present the threats to validity and Section VII concludes our study.

II. Related Work

The work that is most related to ours comes from two areas, work related that uses source-code comments and work in the area of technical debt.

Work using source-code comments: Previous work on source-code comments focused on studying comment updates [6], [8], [10], [11] and how comments can assist in task assignment and task completion [9], [12], [13].

Fluri *et al.* [8] examined the co-evolution of code and comments in ArgoUML, Azureus, and JDT Core and found that new code is rarely commented and that 97% of comment changes are done in the same revision as the code (i.e., consistently co-changed). Malik *et al.* [10] propose techniques to predict the likelihood of a comment being updated and were able to do so with high accuracy (i.e., 80%). Tan *et al.* [6] analyzed inconsistencies in code and corresponding comments by classifying comments as either those that would lead to bugs and comments that do not sync with the source code. They propose a tool called iComment and show that iComment can achieve an accuracy between 90.8-100%. In follow-on work, Tan *et al.* [11] studied the inconsistencies between method bodies and Javadoc comments.

Other work used comments to assist in task assignment and completion. Storey *et al.* [9] investigated how task annotations can assist developers achieve tasks and enhance communication about the code. They found that the use of task annotations varies from individuals to teams and if incorrectly managed, could negatively impact the maintenance of a system. Khamis *et al.* [12] focus on assessing the quality of documentation in software and need for maintaining source-code comments for better documentation. Padioleau *et al.* [13] manually examined 1050 comments from Linux, FreeBSD and OpenSolaris and found that 52.6% of the examined comments can be leveraged by tools to improve reliability.

[1]http://users.encs.concordia.ca/ eshihab/data/ICSME2014/satd.html

Our work differs from the prior work in several ways. First, the main focus of our work is on self-admitted technical debt. In particular, we focus on quantifying and understanding why and how much self-admitted technical debt is removed after its introduction, in contrast to the prior work which focused on the inconsistency on code and comment co-change or the examination of source-code comments to assist in task completion. Second, our work examines the personnel aspect (i.e., developers involved) of self-admitted technical debt, which is something prior work did not investigate. Lastly, our work also complements the prior work since we also examine the co-change of code and comments in Section V in order to shed light on the validity of our approach. Our findings corroborate the findings of the prior work, especially the work in [8].

Work on technical debt: Zazworka *et al.* [14] identified technical debt by asking a project team to identify technical debt items in artifacts from their software project. The goal of the study was to ascertain if automated tools find the same technical debt items as the project team and quantify the overlap of items. The technical debt items identified by both approaches were categorized under defect debt, design debt, documentation debt, testing debt and usability debt. Kruchten *et al.* [15] present an in-depth description of technical debt. The study notes how organizations have embraced technical debt, by making technical debt visible, integrating it into planning and considering it into future risks. The study concludes that poorly managed risks can negatively affect the future of the software but properly managed risks can add value to the software "in the form of deferred investment opportunities". Guo *et al.* [16] analyzed the effect of technical debt by tracking a single delayed task in a software project throughout its lifecycle.

In many ways, our work complements the aforementioned work on technical debt. First, our work focuses on self-admitted technical debt, which can be thought of as intentional technical debt. Second, our work performs an exploratory study on self-admitted technical debt, hence, contributing to the body of work on technical debt. Lastly, since we share our data from this study, we believe that sharing this data will help spur new research in the area of self-admitted technical debt and technical debt in general.

III. Case-Study Setup

The goal of this study is to better understand self-admitted technical debt. In particular, we focus on quantifying the amount self-admitted technical debt, examining why self-admitted technical debt is introduced and determining the amount of self-admitted technical debt that is removed after their introduction. We formalize our study in the following research questions:

RQ1. How much self-admitted technical debt exists in the studied software projects?

RQ2. Why is self-admitted technical debt introduced into the software project?

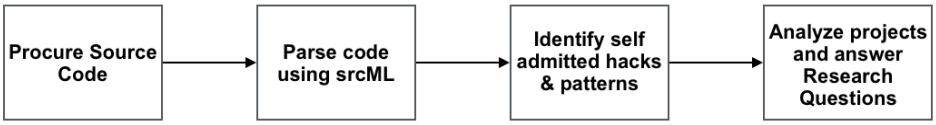

Fig. 1. Approach Overview

TABLE I
PROJECT STATISTICS

Project	Release	Release Date	Code Lines	Comment Lines	Committers
Eclipse	4.3	June 2013	659,231	437,640	221
Chromium OS	30	November 2009	107,706	37,889	1,784
ArgoUML	0.34	December 2011	122,575	115,713	53
Apache httpd	2.4.6	July 2913	192,333	54,295	145

RQ3. How much self-admitted technical debt is removed after its introduction?

To conduct our study, we use data from four, large open-source projects namely - Eclipse, Chromium OS, Apache httpd, and ArgoUML. When selecting our case-study projects, we wanted to have projects that are long-lived, have a large number of contributors and are well commented (since a large part of our analysis depends on the comments). We selected Eclipse, Chromium OS, Apache httpd, and ArgoUML for our study since they have large and accessible codebases with a lengthy development history. Furthermore, `Ohloh.net` mentions that the projects have "a well established, mature codebase" and "average" to "well-commented source code".

Figure 1 provides an overview of the approach used to perform our study. First, we mine the source code for the four open source projects. Then, we parse the code to distinguish comment lines and source code. We read through the comments to identify patterns in the comments that indicate self-admitted technical debt. Then, we use these identified patterns to automatically identify self-admitted technical debt in the projects. Lastly, we use the self-admitted technical debt to perform our analysis and answer our research questions.

A. Extracting Repository Data

To perform our analysis, we require the source code as input. We extracted the latest public releases available at the time from each project's source control systems. To assure validity of our results, we extracted data for four large open source projects - namely Eclipse, Chromium OS, ArgoUML and Apache httpd. Once we extracted the files, we found that Chromium OS files contained both Java and C++ files. Chromium OS constituted of 544 Java files and 13,784 C++ files; ArgoUML contained 1,846 Java files; Apache HTTP Server had 255 C files; Eclipse had 6,389 Java files. Table I represents statistics for all the studied projects.

The source code extracted for each project was the latest release before January 2014, listed in Table I. While Eclipse, Apache httpd, and ArgoUML had public releases closer to the date of extraction, the only public release by Chromium OS was a few years prior. Chromium OS uses different release channels to roll out stable updates to users roughly every six weeks [17]. Hence, for Chromium OS, we extracted the latest source code available on their repository which corresponded to branch point 30.

B. Parsing Code Using srcML

After obtaining the source code of the software projects, we needed to discriminate between source code and comment lines. To extract the source-code comments, we utilized the srcML Toolkit [18], a command line tool that parses source code into XML files. Then, we developed a Java-based tool that parses the XML files produced by srcML, and stores the comment nodes of the code into separate text files for each project. The Chromium OS Java files contained 12,387 comments while the C++ files contained 3,944 comments. The Eclipse, ArgoUML and Apache projects contained 9,273, 62,256, and 13,902 comments, respectively. The srcML tool reads a comment block - a comment that spans multiple lines - as a single comment and hence it is important to note that the aforementioned numbers denote the number of comments and not the number of lines of comments in the source files.

C. Identifying self-admitted Technical Debt

Once the comments are extracted, our next step is to determine the comments that indicate self-admitted technical debt. Since comments are written in natural language, it is very difficult to automatically analyze them. Therefore, the comments were analyzed manually by reading through each of them to identify those that indicated self-admitted technical debt. In total, Potdar read through 101,762 comments. A link containing the all of the comments analyzed is provided[2]. Table II provides a sample of comments that were identified as indicating self-admitted technical debt.

D. Identifying Specific Self-Admitted Technical-Debt Patterns

Once we identified all of the comments that indicate self-admitted technical debt, we needed to further distill these comments to specific patterns that indicate self-admitted technical debt (e.g., we removed stop words). In total, we ended up with a set of 62 recurring patterns that were identified across

[2]http://users.encs.concordia.ca/ eshihab/data/ICSME2014/satd.html

TABLE II
SAMPLE COMMENTS INDICATING SELF-ADMITTED TECHNICAL DEBT

Project	Comment
Eclipse	// TODO this is such a hack it is silly
Chromium OS	// Unsafe; should error.
ArgoUML	// FIXME: This is such a gross hack...
Apache	/* Ugly, but what else? */

TABLE III
SELF-ADMITTED TECHNICAL-DEBT COMMENT INSTANCES

Type	Project	Total	Self-Admitted Technical Debt	%
Files	Eclipse	6,389	152	2.4%
	Chromium OS	14,328	574	4.0%
	ArgoUML	1,846	77	4.2%
	Apache httpd	255	79	31.0%
Classes	Eclipse	11,993	151	1.3%
	Chromium OS	18,249	82	0.4%
	ArgoUML	2,304	76	3.3%
	Apache httpd	-	-	-%
Methods/ Functions	Eclipse	48,382	145	0.3%
	Chromium OS	153,514	626	0.4%
	ArgoUML	12,845	71	0.6%
	Apache httpd	4,361	112	2.6%
Constructors	Eclipse	5,561	151	0.2%
	Chromium OS	18,988	82	0.1%
	ArgoUML	1,920	76	0.5%
	Apache httpd	-	-	-%

the four projects. The following are examples of patterns we identified: *hack*, *fixme*, *is problematic*, *this isn't very solid*, *probably a bug*, *hope everything will work*, *fix this crap*. A full list of the patterns in provided in the supplementary data[3]. We use the final set of 62 patterns to perform all of our analysis, which we discuss in the next section.

IV. CASE-STUDY RESULTS

The goal of our research is to perform an exploratory study on self-admitted technical debt. We study data from four, large open source projects namely - Eclipse, Chromium OS, Apache httpd, and ArgoUML. Since, to the best of our knowledge, this is one of the first studies to focus on self-admitted technical debt, our first question is related to the quantity of self-admitted technical debt.

RQ1. How much self-admitted technical debt exists in the studied software projects?

Motivation: Intuitively, self-admitted technical debt is not considered to be good practice. At the same time, our experience indicates that it does exist [19]. However, how common self-admitted technical debt is is still unknown. Quantifying the amount of self-admitted technical debt that exists helps us better understand how much of a problem it poses and whether it warrants more attention or not.

Approach: To quantify how much self-admitted technical debt exists in the different projects, we use the 62 self-admitted technical-debt patterns described earlier in Section III. For each source-code file, we quantified the number of self-admitted technical debt that exists using their comments. Since different projects have a different number of comments, we normalized our results by the total number of comments that the project has. For example, if a project has 100 comments in total and another has 1,000 comments in total, and 10 comments in each project match one of the 62 patterns identified earlier, then we say that 10% of the first project and 1% of the second project contains self-admitted technical debt. Since comments can be mapped at different granularities, we report the results of our analysis at the file, class and method/function level.

Results: Table III shows the results of our analysis at different levels of granularity. We find that at the *file* level, between 2.4 - 31.0% of the files contained one or more instances of self-admitted technical debt. Since source-code comments and therefore self-admitted technical debt also lie outside the boundary of a class, as expected, the percentage was lower for the *class* level, ranging between 0.4 - 3.3% and 0.3 - 2.6% at the *method/function* level. Table III also shows the raw number of files, classes and methods/functions. The class analysis does not include results for the Apache httpd project since Apache httpd is written in C, and therefore, it does not have classes.

For Eclipse and ArgoUML a couple of anomalies were noted where the number of files with self-admitted technical debt outnumbered the number of classes with self-admitted technical debt. On further analysis, we found that there existed self-admitted technical debt in the files that were located outside the Class constructs, i.e., in constructor declarations. Parsing the source code with srcML places the constructor nodes outside the scope of the class nodes. Therefore, we also include the number and percentage of self-admitted technical debt at the bottom of Table III.

To shed light on the most frequent self-admitted technical debt, we also quantified the most recurring patterns across all of the projects. Moreover, to determine whether these patterns were all by a single developer or multiple developers, we also measure the number of unique developers that use a pattern.

We found that in Eclipse, the most commonly occurring self-admitted technical debt pattern is *there is a problem* with 36 instances by 10 different developers followed by *workaround for bug* with 30 instances by 7 different developers. In ArgoUML, the most commonly occurring pattern was *hack* with 17 instances by 4 different developers followed by *give up* with 14 instances by 2 different developers. In Apache httpd, the most commonly occurring pattern was *fixme* with 20 instances by 17 different developers followed by *kludge* with 19 instances by 6 different developers. In Chromium OS, the most commonly occurring pattern was *fixme* with 761 instances by 212 different developers followed by *hack* with 89 instances by 35 different developers. This finding clearly shows that self-admitted technical debt is not only by a single developer, but is actually made by different developers.

[3]http://users.encs.concordia.ca/ eshihab/data/ICSME2014/satd.html

We find that 2.4 - 31.0% of files, 0.4 - 3.3% of classes and 0.3 - 2.6% of methods/functions contain self-admitted technical debt.

RQ2. Why is self-admitted technical debt introduced into the software project?

Motivation: After quantifying the amount of self-admitted technical debt, we want to better understand why self-admitted technical debt is introduced. To answer this research question, we investigate three types of factors that might be causing the introduction of self-admitted technical debt: *experience, time and complexity*. Our conjuncture is that less experienced developers do not know the codebase very well, and therefore, introduce self-admitted technical debt. For time factors, our conjuncture is that developers are pressured by tight release deadlines and are therefore introducing self-admitted technical debt close to the release dates. Finally, for the complexity factors, our conjuncture is that more complex code is harder to change, therefore, developers are more likely to introduce self-admitted technical debt. Answering this question will help us and future research better address this issue of self-admitted technical debt.

Approach: To investigate whether or not **experience** plays a role in the introduction of self-admitted technical debt, we used the *git blame* (for Eclipse and Chromium OS) and *svn blame* (for ArgoUML and Apache) commands to know which developer introduced the self-admitted technical debt. Then, to get the developer's experience on the project, we measured the number of commits the developer made prior to the commit that introduced the self-admitted technical debt. The number of prior commits serves as a proxy for experience and has been used in a similar manner in previous work [20], [21].

We also measure when in their development experience developers introduce self-admitted technical debt. To do so, we obtained the timestamps of all commits. Then, we used the obtained timestamp to build scatter plots that superimpose the self-admitted technical debt over all the commits that a developer makes.

To investigate whether or not **time** pressure plays a role in the introduction of self-admitted technical debt, we quantify the number of self-admitted technical debt introduced more than 6 months before a release, between 3 and 6 months before a release, between 1 and 3 months and within less than a month before a release.

To investigate whether or not **complexity** plays a role in the introduction of self-admitted technical debt, we measure the Spearman correlation between the dependencies (i.e., Fan-in) of files and the number of self-admitted technical debt they contain. Since larger files have more commits overall and will naturally have a higher likelihood of having more self-admitted technical debt, we measured the partial correlations (rather than the simple Spearman correlations) between the dependencies and the number of self-admitted technical debt.

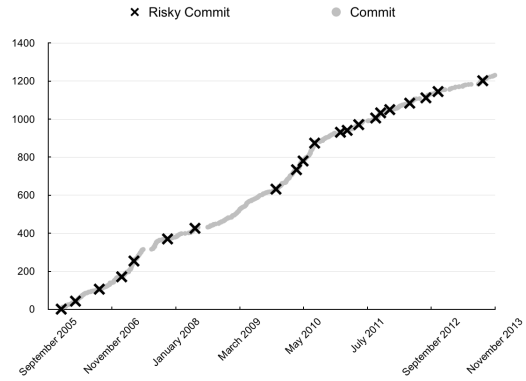

Fig. 2. Eclipse: Commits by *E1*

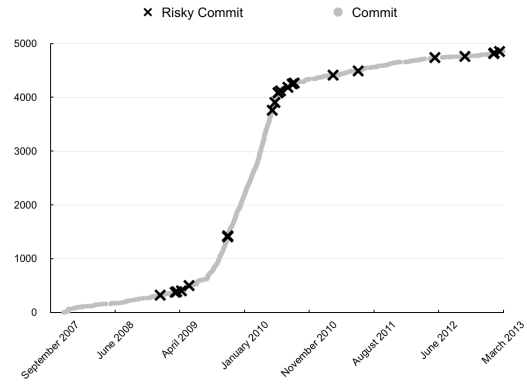

Fig. 3. Chromium OS: Commits by *C1*

The partial correlation measures the degree of association between two variables, while removing the effect of a controlling variable [22]. In our case, size of a file, measured in LOC, is used as the controlling variable.

Results - Developer Experience: Table IV shows the top 5 contributors of self-admitted technical debt in the four studied projects. For each contributor, we provide their total number of commits, the number of self-admitted technical debt and their rank in terms of experience. We find that in Eclipse and ArgoUML, the top 5 contributors in terms of the number of self-admitted technical debt are in the top 4% $(100 * \frac{9}{221})$ and 15% $(100 * \frac{8}{53})$ of most experienced contributors, respectively.

In Apache httpd, the top 5 contributors of self-admitted technical debt are ranked in the top 18% in terms of experience. However, the outlier here is the Chromium OS, where we find that the most experienced contributor is also the one with the most self-admitted technical debt, however, there are other contributors such as `C2` and `C5` who are ranked much lower. At first, the fact that someone with an experience rank as low as 232 is in the top 5 contributors to self-admitted technical debt seems out of place. However, given that Chromium OS also has a large number of contributors, we find that Chromium OS is similar to the other projects, i.e., the top 5 contributors of self-admitted technical debt are ranked in the top 13% $(100 * \frac{232}{1784})$ in terms of experience.

In addition, to see when in their experience developers

TABLE IV
SELF-ADMITTED DEBT-COMMITS BY DEVELOPERS

Project	Developer	Total Commits	Self-Admitted Technical Debt	Experience Rank	Total Committers
Eclipse	E1	1,231	21	8	
	E2	2,513	20	2	
	E3	1,508	12	5	221
	E4	4,436	10	1	
	E5	1,190	7	9	
Chromium OS	C1	4,853	28	1	
	C2	410	24	219	
	C3	1,034	21	64	1,784
	C4	1,930	21	26	
	C5	388	21	232	
ArgoUML	AU1	2,544	17	4	
	AU2	4,879	7	1	
	AU3	3,088	6	3	53
	AU4	3,192	4	2	
	AU5	480	3	8	
Apache	AH1	4,908	14	1	
	AH2	3,214	13	3	
	AH3	721	11	26	145
	AH4	1,219	11	13	
	AH5	1,239	10	11	

introduce self-admitted technical debt, we plotted the commits for the top developer, in terms of their experience, of each project. Figures 2, 3, 4, and 5 show the commits for the top developers of the Eclipse Platform, Chromium OS, ArgoUML and Apache httpd, respectively. The gray line shows the cumulative number of commits by the developer and the 'x' indicates a self-admitted technical debt. In all cases, we observe that developers introduce self-admitted technical debt throughout their development experience, i.e., not only at the beginning of their development or later in their development experience.

> Developers with higher experience tend to introduce more self-admitted technical debt into software projects. At the same time, developers introduce self-admitted technical debt throughout their project experience.

Results - Time to release: Table V shows the number of self-admitted technical debt, the total number of changes, the ratio of self-admitted technical debt to total changes, and the ratio of self-admitted technical debt to the total number of self-admitted technical debt at different times before a release for Eclipse, ArgoUML and Apache httpd. Note that we do not include the results for Chromium OS, since there has been only one public release of the project and hence we would not be able to derive a comparative analysis.

Initially, we measured the raw number of self-admitted technical debt at different time intervals before a release, however, we realized that the number of total changes will vary at different time intervals before a release as well. Therefore, we report the ratio of self-admitted technical debt to total changes and the ratio of self-admitted technical debt to the total number of self-admitted technical debt. We find that in all of the studied projects, most of the self-admitted technical debt is introduced more than 3 months before a release. In

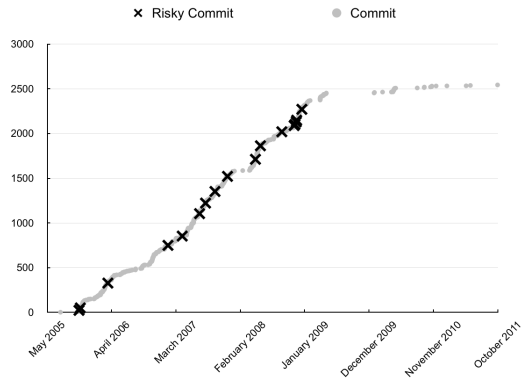

Fig. 4. ArgoUML: Commits by *AU1*

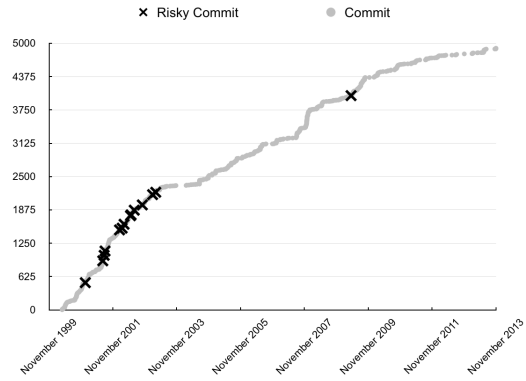

Fig. 5. Apache: Commits by *AH1*

fact, in all the studied projects, less than 15% of the self-admitted technical debt is introduced one month before the release. This finding leads us to the conclusion that in our case-study projects, release pressure does not play a major role in the introduction of self-admitted technical debt.

TABLE V
DEBT COMMITS - TIME TO RELEASE WINDOWS

Project	Time to Release	Self-Admitted Technical Debt	Total Changes	% Over Total Changes	% Self-Admitted Technical Debt
Eclipse	>= 6month	55	16,682	0.33%	41.7%
	>= 3month & <= 6month	48	7,452	0.64%	36.4%
	>= 1month & <= 3month	22	5,591	0.39%	16.7%
	<= 1month	7	1,777	0.39%	5.3%
ArgoUML	>= 6month	16	7,973	0.20%	33.3%
	>= 3month & <= 6month	20	5,917	0.34%	41.7%
	>= 1month & <= 3month	8	4,315	0.19%	16.7%
	<= 1month	4	1,708	0.23%	8.3%
Apache	>= 6month	51	23,579	0.22%	37.5%
	>= 3month & <= 6month	43	13,960	0.31%	31.6%
	>= 1month & <= 3month	28	11,670	0.24%	20.6%
	<= 1month	14	6,614	0.21%	10.3%

TABLE VI
PARTIAL SPEARMAN CORRELATIONS BETWEEN CYCLOMATIC COMPLEXITY, FAN-IN, FAN-OUT AND THE NO. OF SELF-ADMITTED TECHNICAL DEBT WHILE CONTROLLING FOR SIZE

Project	Cyclomatic Comp.	Fan-in	Fan-out
Eclipse	-0.1992618 **	0.3159485	-0.08568448
Chromium OS	0.1840126 **	-0.01470615 *	-0.2529673 *
ArgoUML	-0.03449873 *	-0.01459957 *	0.1686399
Apache	0.2386505 **	0.3333192 ***	0.1108314 *

($p < 0.01$ ***; $p < 0.1$ **; $p < 1$ *)

Release pressure does not play a major role in the introduction of self-admitted technical debt. Less than 15% of the self-admitted technical debt is introduced within 1 month of the latest release.

Results - Complexity: Table VI shows the partial correlations between the number of self-admitted technical debt in a file and the McCabe cyclomatic complexity, fan-in and fan-out of the file, while controlling for size. We also indicate the p-value of the partial correlations next to the correlation value in the table.

From Table VI, we find that for all projects, there is weak positive or weak negative correlation between the complexity metrics and self-admitted technical debt. Although this finding is not conclusive, the fact that all the correlation values are low indicates that, at least in terms of correlations, self-admitted technical debt is not correlated with any complexity metrics. It is important to note here that our finding is based on correlations and in no way indicate a causation.

When controlling for size, we find a weak to very weak correlation between the number of self-admitted technical debt in a file and complexity.

TABLE VII
DEBT PATTERNS ACROSS ECLIPSE RELEASES

Version	Release Date	Removed Self-Admitted Technical Debt	Change
3.0 − > 3.0	June 2004	-	-
3.0 − > 3.1	June 2005	26.3%	26.3%
3.0 − > 3.2	June 2006	27.9%	1.6%
3.0 − > 3.3	June 2007	29.9%	2%
3.0 − > 3.4	June 2008	35.6%	5.7%
3.0 − > 3.5	June 2009	35.6%	0%
3.0 − > 3.6	June 2010	36.9%	1.3%
3.0 − > 3.7	June 2011	36.9%	0%

TABLE VIII
DEBT PATTERNS ACROSS ARGOUML RELEASES

Version	Release Date	Removed Self-Admitted Technical Debt	Change
0.20 − > 0.20	February 2006	-	-
0.20 − > 0.22	August 2006	37.3%	37.3%
0.20 − > 0.24	February 2007	41.3%	4.0%
0.20 − > 0.26	September 2008	58.7%	17.4%
0.20 − > 0.28	March 2009	62.7%	4.0%
0.20 − > 0.30	March 2010	65.3%	2.6%
0.20 − > 0.32	January 2011	65.3%	0%
0.20 − > 0.34	December 2011	65.3%	0%

RQ3. How much self-admitted technical debt is removed after its introduction?

Motivation: In the majority of cases, when self-admitted technical debt is introduced, they are meant to be removed or addressed later on. However, whether this self-admitted technical debt is indeed removed and exactly how much of this self-admitted technical debt is removed is not known. Therefore, in this research question, we quantify the amount of self-admitted technical debt that is removed after its introduction.

Approach: To determine how much self-admitted technical debt is removed, we compute how much of the self-admitted technical debt that was introduced remains in future releases.

TABLE IX
DEBT PATTERNS ACROSS APACHE RELEASES

Version	Release Date	Removed Self-Admitted Technical Debt	Change
1.3.x − > 1.3.x	February 2002	-	-
1.3.x − > 2.0.x	January 2004	65.45%	65.45%
1.3.x − > 2.2.x	December 2005	73.64%	8.19%
1.3.x − > 2.4.x	February 2012	74.55%	0.91%

First, we identify all the self-admitted technical debt in the first release of the project. Then, we search for each of the identified self-admitted technical debt in the future releases. It is important to note that we used the comments in the following releases to identify the self-admitted technical debt. We discuss the implications of using this approach later in Section V. Since we could only identify a single public release of Chromium OS, we were unable to perform this analysis for Chromium OS.

Results: We present our results on a per project basis. Table VII shows the results for *Eclipse*. We started by identifying all of the 301 instances of self-admitted technical debt in release 3.0 and then measuring how much of this remained in the following releases. For Table VII, we see that 26.3% of the self-admitted technical debt found in release 3.0 was removed in release 3.1. Even after 7 releases, approximately 63% of the self-admitted technical debt remained in Eclipse. Furthermore, we see from Table VII that the majority of the self-admitted technical debt is removed after the first release.

Table VIII shows the results for *ArgoUML*. Release 0.20 of ArgoUML had 75 instances of self-admitted technical debt. Of that, 37.3% was removed in the immediate next release. After seven releases, 65.3% of the self-admitted technical debt was removed. In contrast to Eclipse, in ArgoUML, most of the self-admitted technical debt was removed. However, similar to Eclipse, the majority of the self-admitted technical debt was removed after the first release.

Table IX shows the results for *Apache httpd*. Release 1.3.0 had a total of 110 instances of self-admitted technical debt. After four major releases (in 10 years), 25.45% of the self-admitted technical debt remains in the system. Similar to ArgoUML, we observe that the majority of self-admitted technical debt is removed. Furthermore, similar to both Eclipse and ArgoUML, most of the self-admitted technical debt is removed after one release.

> We find that between 26.25% to 63.45% of the self-admitted technical debt is removed in following releases. The majority of the self-admitted technical debt is removed in the immediate next release, however, self-admitted technical debt tends to persist in software projects over multiple releases.

TABLE X
SCENARIO INSTANCES SUMMARY

Release	Case	Code	Comment	Count
R3.0 − > R3.1	1	Changed	Changed	57
R3.0 − > R3.1	2	Changed	Unchanged	14
R3.0 − > R3.1	3	Unchanged	Changed	11
R3.0 − > R3.1	4	Unchanged	Unchanged	203
R3.0 − > R3.7	1	Changed	Changed	73
R3.0 − > R3.7	2	Changed	Unchanged	64
R3.0 − > R3.7	3	Unchanged	Changed	10
R3.0 − > R3.7	4	Unchanged	Unchanged	122

V. DISCUSSION

In this paper, we answer questions related to how much, by who, when, and why self-admitted technical debt exist. To identify self-admitted technical debt, we used the comments provided by the developers themselves. For example, in RQ3, we used the comments to investigate how much self-admitted technical debt is removed in the future. However, if a developer updates the code without updating the comment, then we may wrongly assume that the self-admitted technical debt still exists, where in reality it does not.

Therefore, in this section, we quantify the scenarios where code and comments are updated inconsistently [11]. In particular, we consider the following four cases:

Case1. The self-admitted technical debt was removed along with change in enclosing code

Case2. The self-admitted technical debt was removed but enclosing code was unchanged

Case3. The self-admitted technical debt persisted despite enclosing code changing

Case4. The self-admitted technical debt persisted with no change in enclosing code

In the listed cases, Case 1 and Case 4 are considered to be consistent updates, i.e., the code and comments were both updated or neither the code nor the comment were updated. Case 2 and Case 3 are the two problematic cases, i.e., where the code was updated but not the code and vice versa. To verify whether using the comments to identify self-admitted technical debt is a viable approach, we are mainly interested in quantifying the frequency of Cases 2 and 3.

To quantify the amount of inconsistent changes, we select the first release from each project and use that as the base version. For the base version of each project, we measure the number of self-admitted technical debt in each file. Then, we locate the file in future releases and subsequently match the comment and code to determine the various cases. We developed a tool that navigates through the files of the base version, identify a file containing the self-admitted technical debt, search for the file in the following releases and make both files easily available for us to analyze. Once the files were procured, we manually compared the code and comments of the files of the base version against the files in the future releases and categorized them under one of the four above mentioned cases. Due to space limitations, we perform this

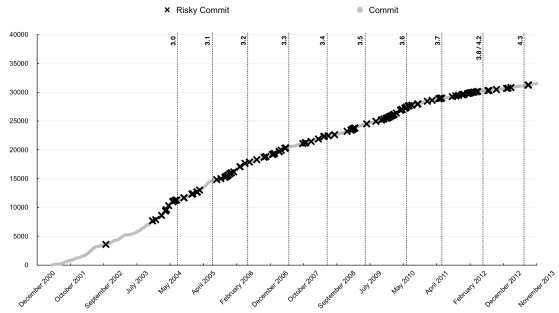

Fig. 6. Eclipse Risky Commits

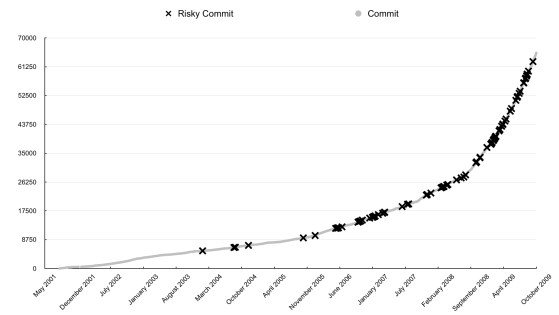

Fig. 7. Chromium OS Risky Commits

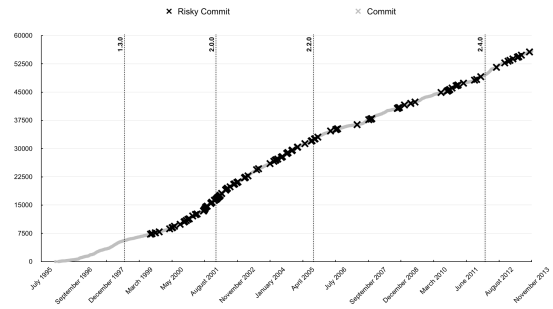

Fig. 8. Apache Risky Commits

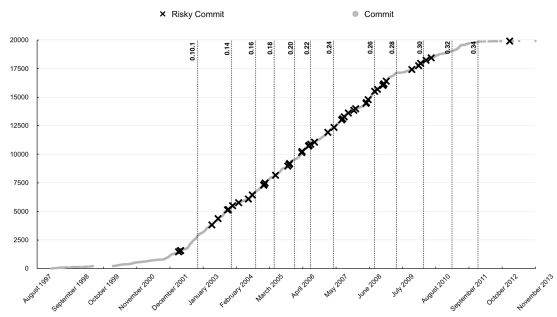

Fig. 9. ArgoUML Risky Commits

experiment on the Eclipse software project. We use version 3.0 (R3.0), released in June 2004, as the base release. We investigate by comparing against two following releases - version 3.1 (R3.1), released in June 2005 and version 3.7 (R3.7), released in June 2011.

Table X shows our findings for each case in releases R3.1 and R3.7. From the Table X, we observe that in R3.1 approximately 8.8% ($\frac{14+11}{57+14+11+203}$) of the self-admitted technical debt is inconsistently changed (i.e., belong to cases 2 and 3). For R3.7, approximately 27.5% ($\frac{60+10}{73+64+10+122}$) are inconsistently changed. One possible reason for the increase in inconsistent changes is that as the time between base release and the actual release increases, developers forget that a self-admitted technical-debt comment existed and just update the code or update the comment without changing the code since it might have been addressed in another way. In any case, our findings show that the majority of the self-admitted technical debt is consistently changed (i.e., either the code and comments change consistently or the code and comments both do not change).

In this paper, we look at who introduces self-admitted technical debt but it would also be interesting to further delve into why less experienced developers introduce less technical debt, or if at all they simply do not admit it. Since most technical debt is never removed and introduced without any time pressure, should developers be taking specific precautions to avoid self-admitted technical debt. These further questions can be part of future work on this topic.

VI. Threat to Validity

In this section, we discuss the possible threats to the validity of our study:

Threats to Internal Validity refers specifically to whether an experimental condition makes a difference or not, and whether there is sufficient evidence to support the claim of the study. To identify self-admitted technical debt, we needed to identify the comments that would distinguish such self-admitted technical debt. Since comments are written in natural language, they had to be analyzed manually to identify those that would indicate self-admitted technical debt. Any manual process is prone to human error and/or subjectivity. On the same point, as mentioned in our discussion section (Section V), in some cases, using the comments to determine some self-admitted technical debt may not be fully representative since comments or code may not be updated consistently. As shown in Section V, most of the self-admitted technical debt is updated consistently. Furthermore, it is important to note that our work focuses on *self-admitted* technical debt and not all technical debt. There may exist a variety of technical debt that is not self-admitted. Considering all technical debt is out of the scope of this work.

Through manual examination, we identified a set of 62 recurring patterns that we use to determine self-admitted technical debt. We built this set by studying source-code comments of only the four projects we selected. Our choice of recurring patterns may have an impact on our findings. To help alleviate this threat, we manually examined each comment and performed this step using four different projects.

Threats to External Validity refers to the generalizability of the outcomes of the study. Our study uses four large, well-established open source software projects with well commented source code. Our results may not necessarily generalize to all other open source or commercial projects. A large part of our analysis depends on source-code patterns detected from the four studied projects. Hence, these patterns may be different for different projects.

VII. Conclusion

Developers knowingly commit code that is either incomplete, requires rework, produces errors, or is a temporary workaround. These temporary fixes are often referred to as technical debt, and in many cases, developers admit when they are coding such technical debt. We call such debt self-admitted technical debt. Although such self-admitted technical debt is common, very little empirical evidence is known about self-admitted technical debt. Therefore, in this paper we perform an exploratory study, using four large open source projects, to determine how much self-admitted technical debt code exists, why self-admitted technical debt exists and how much of this self-admitted technical debt is removed after its introduction. Our findings show that 2.4 - 31.0% of the files in a project contain self-admitted technical debt, that developers with higher experience tend to introduce more self-admitted technical debt and that time pressure and complexity do not correlate with self-admitted technical debt. Finally, we find that even after multiple releases, only between 26.3 - 63.5% of the self-admitted technical debt is removed after its introduction.

Our findings shed light on the extent and existence of self-admitted technical debt and we plan (and hope that others) will use this study as motivation to dedicate more work on this important area of software maintenance. The findings also serve as motivation to build tools that support developers so they can avoid self-admitted technical debt or at least help them track and manage such self-admitted technical debt after its introduction.

References

[1] Erin Lim, Nitin Taksande, and Carolyn Seaman. A balancing act: What software practitioners have to say about technical debt. *IEEE Softw.*, 29(6):22–27, November 2012.

[2] Nico Zazworka, Michele A. Shaw, Forrest Shull, and Carolyn Seaman. Investigating the impact of design debt on software quality. In *Proceedings of the 2Nd Workshop on Managing Technical Debt*, MTD '11, pages 17–23, New York, NY, USA, 2011. ACM.

[3] Thomas Zimmermann, Nachiappan Nagappan, and Andreas Zeller. *Software Evolution*, chapter Predicting Bugs from History, pages 69–88. Springer, 2008.

[4] Kai Pan, Sunghun Kim, and E. James Whitehead, Jr. Toward an understanding of bug fix patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[5] Bee Bee Chua. Rework requirement changes in software maintenance. In *Proceedings of the 2010 Fifth International Conference on Software Engineering Advances*, ICSEA '10, pages 252–258, Washington, DC, USA, 2010. IEEE Computer Society.

[6] Lin Tan, Ding Yuan, Gopal Krishna, and Yuanyuan Zhou. /*icomment: Bugs or bad comments?*/. *SIGOPS Oper. Syst. Rev.*, 41(6):145–158, October 2007.

[7] Zhen Ming Jiang and Ahmed 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, New York, NY, USA, 2006. ACM.

[8] Beat Fluri, Michael Wursch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, Washington, DC, USA, 2007. IEEE Computer Society.

[9] Margaret-Anne Storey, Jody Ryall, R. Ian Bull, Del Myers, and Janice Singer. Todo or to bug: Exploring how task annotations play a role in the work practices of software developers. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 251–260, New York, NY, USA, 2008. ACM.

[10] Haroon Malik, Istehad Chowdhury, Hsiao-Ming Tsou, Zhen Ming Jiang, and Ahmed E. Hassan. Understanding the rationale for updating a function's comment. In *ICSM*, pages 167–176. IEEE, 2008.

[11] Shin Hwei Tan, Darko Marinov, Lin Tan, and Gary T. Leavens. @tcomment: Testing javadoc comments to detect comment-code inconsistencies. In *Proceedings of the 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, ICST '12, pages 260–269, Washington, DC, USA, 2012. IEEE Computer Society.

[12] Ninus Khamis, René Witte, and Juergen Rilling. Automatic quality assessment of source code comments: The javadocminer. In *Proceedings of the Natural Language Processing and Information Systems, and 15th International Conference on Applications of Natural Language to Information Systems*, NLDB'10, pages 68–79, Berlin, Heidelberg, 2010. Springer-Verlag.

[13] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. Listening to programmers taxonomies and characteristics of comments in operating system code. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 331–341, Washington, DC, USA, 2009. IEEE Computer Society.

[14] Nico Zazworka, Rodrigo O. Spínola, Antonio Vetro', Forrest Shull, and Carolyn Seaman. A case study on effectively identifying technical debt. In *Proceedings of the 17th International Conferenceon Evaluation and Assessment in Software Engineering*, EASE '13, pages 42–47, New York, NY, USA, 2013. ACM.

[15] Philippe Kruchten, Robert L. Nord, Ipek Ozkaya, and Davide Falessi. Technical debt: Towards a crisper definition report on the 4th international workshop on managing technical debt. *SIGSOFT Softw. Eng. Notes*, 38(5):51–54, August 2013.

[16] Yuepu Guo, Carolyn Seaman, Rebeka Gomes, Antonio Cavalcanti, Graziela Tonin, Fabio Q. B. Da Silva, Andre L. M. Santos, and Clauirton Siebra. Tracking technical debt – an exploratory case study. In *Proceedings of the 2011 27th IEEE International Conference on Software Maintenance*, ICSM '11, pages 528–531, Washington, DC, USA, 2011. IEEE Computer Society.

[17] Google. Chrome release channels.

[18] Michael L. Collard, Michael John Decker, and Jonathan I. Maletic. Lightweight transformation and fact extraction with the srcml toolkit. In *SCAM*, pages 173–184. IEEE, 2011.

[19] Eric Allman. Managing technical debt. *Commun. ACM*, 55(5):50–55, May 2012.

[20] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: Examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 4–14, New York, NY, USA, 2011. ACM.

[21] Emad Shihab, Ahmed E. Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 62:1–62:11, New York, NY, USA, 2012. ACM.

[22] SAS. Base sas(r) 9.2 procedures guide: Statistical procedures, third edition.