Detecting and Quantifying Different Types of Self-Admitted Technical Debt

Everton da S. Maldonado and Emad Shihab
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada
{e_silvam,eshihab}@encs.concordia.ca

***Abstract*—Technical Debt is a term that has been used to express non-optimal solutions during the development of software projects. These non optimal solutions are often shortcuts that allow the project to move faster in the short term, at the cost of increased maintenance in the future. To help alleviate the impact of technical debt, a number of studies focused on the detection of technical debt. More recently, our work shown that one possible source to detect technical debt is using source code comments, also referred to as self-admitted technical debt. However, what types of technical debt can be detected using source code comments remains as an open question.**

Therefore, in this paper we examine code comments to determine the different types of technical debt. First, we propose four simple filtering heuristics to eliminate comments that are not likely to contain technical debt. Second, we read through more than 33K comments, and we find that self-admitted technical debt can be classified into five main types - design debt, defect debt, documentation debt, requirement debt and test debt. The most common type of self-admitted technical debt is design debt, making up between 42% to 84% of the classified comments. Lastly, we make the classified dataset of more than 33K comments publicly available for the community as a way to encourage future research and the evolution of the technical debt landscape.

I. Introduction

The software development process is filled with challenges. There are short deadlines, complex changes that need to be made, high quality expectations and an ever changing environment. Often there is much more that needs to be done than time to accomplish it. These conditions puts developers under increasing pressure to implement their tasks, while achieving many conflicting constraints. In this context, some decisions are made to allow the short term development of the project at the cost of its increased maintenance effort in the future. This phenomena is know as Technical Debt [1].

With the organization of the technical debt community through the managing technical debt workshop [2], recent work has focused on the detection of technical debt [3], [4], studying the impact of technical debt [5] and the appearance of technical debt in the form of code smells [6]. Despite many efforts to detect technical debt, its detection remains a challenge [3]. One relatively unexplored aspect of technical debt is self-admitted technical debt, that is technical debt reported in source code comments. Self-admitted technical debt refers to the situation where developers know that the current implementation is not optimal and write comments alerting the inadequacy of the solution.

Recently, Potdar and Shihab [3] developed an approach to identify technical debt from code comments, and through manual inspection, were able to mine 62 patterns that effectively identify self-admitted technical debt. However, their approach does not take into consideration the different types of technical debt. Understanding the different types of self-admitted technical debt is important since: 1) it helps the community understand the limitations of understanding technical debt through code comments, 2) it allows us to complement existing technical debt detection approaches and 3) it provides us with a better understanding of the developer's point of view of technical debt.

Therefore, in this paper we examine and quantify the different types of self-admitted technical debt. To do so, we extract source code comments from 5 well commented open source projects that belongs to different application domains, namely Apache Ant, Apache Jmeter, ArgoUml, Columba and JFreeChart. In total, we examined more than 166K comments. We applied a set of 4 simple filtering heuristics to remove comments that are not likely to contain self-admitted technical debt (e.g., license comments, commented source code, Javadoc comments). Finally, these filtering heuristics resulted in a dataset of 33,093 comments that the first author manually analyzed and classified into different types of self-admitted technical debt.

When classifying the code comments, we found 5 types of self-admitted technical debt which are: design debt, defect debt, documentation debt, requirement debt and test debt. Analyzing the distribution of the comments we found that the most common type of self-admitted technical debt is design debt, making up between 42% - 84% of all the classified comments. In addition to our findings, we contribute a rich dataset of self-admitted technical making the data used in this study publicly available [1]. To the best of our knowledge, there is not similar data available and we believe that the dataset will encourage future research in the area of self-admitted technical providing the necessary foundation for more advanced techniques as Natural Language Processing.

The rest of the paper is organized as follows. Section II presents related work. We describe our approach and setup our case study in Section III. Section IV presents the case study results. The threats to validity are presented in Section

[1] http://users.encs.concordia.ca/~e_silvam/publications.html

V and in Section VI concludes the paper and discusses future work.

II. Related Work

Our work uses code comments to classify self-admitted technical debt. Therefore, we divide the related work into two categories: source code comments and technical debt.

A. Source code comments

A number of studies examined the co-evolution of source code comments and the rationale for changing code comments. For example, Fluri *et al.* [7] analyzed the co-evolution of source code and code comments, and found that 97% of the comment changes are consistent. Tan *et al.* [8] proposed a novel approach to identify inconsistencies between Javadoc comments and method signatures. Malik *et al.* [9] studied the likelihood of a comment to be updated and found that call dependencies, control statements, the age of the function containing the comment, and the number of co-changed dependent functions are the most important factors to predict comment updates.

Other work used code comments to understand developer tasks. For example, Storey *et al.* [10] analyzed how task annotations (e.g., TODO, FIXME) play a role in improving team articulation and communication. The work closest to ours is the work by Potdar and Shihab [3], where code comments were used to identify technical debt.

Our work complements the prior work using code comments. Similar to the prior work, we also leverage source code comments, however, we use the comments to identify self-admitted technical debt. In particular, we focus on the detection and quantification of the *different types* of self-admitted technical debt.

B. Technical debt

A number of studies have focused on the study of, detection and management of technical debt. Much of this work has been driven by the Managing Technical Debt Workshop community. For example, Seaman *et al.* [11], Kruchten *et al.* [12], Brown *et al.* [13] and Spinola *et al.* [14] make several reflections about the term technical debt and how it has been used to communicate the issues that developers find in the code in a way that managers can understand. Alves *et al.* [15] proposes an ontology on technical debt terms. In their work they gathered definitions and indicators of technical debt that were scattered across the literature. Their resulting ontology provides several different types of technical debt (e.g., architecture debt, build debt, code debt, design debt, defect debt, etc) grouped by their nature (i.e., the factor that lead to the introduction of the debt at the first place).

Other work focused on the detection of technical debt. Zazworka *et al.* [4] conducted an experiment to compare the efficiency of automated tools in comparison with human elicitation regarding the detection of technical debt. They found that there is small overlap between the two approaches, and thus it is better to combine them than replace one with the other. In addition, they concluded that automated tools are more efficient in finding defect debt, whereas developers can realize more abstract categories of technical debt.

In follow on work, Zazworka *et al.* [5] conducted a study to measure the impact of technical debt on software quality. They focused on a particular kind of design debt, namely God Classes. They found that God Classes are more likely to change, and therefore, have a higher impact in software quality. Fontana *et al.* [6] investigated design technical debt appearing in the form of code smells. They used metrics to find three different 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 a risk scale. Moreover, Potdar and Shihab [3] used code comments to detect self-admitted technical debt.They extracted the comments of four projects and analyzed more than 101,762 comments to come up with 62 patterns that indicates self-admitted technical debt. Their findings show that 2.4% - 31% of the files in a project contain self-admitted technical debt.

Our work is different from the aforementioned work that uses code smells to detect design technical debt since we use code comments to detect technical debt. Also, our focus is on *self-admitted* technical debt. Our work advances the prior work on self-admitted technical debt by detecting and quantifying the different types of self-admitted technical debt and classifying them accordingly. We also contribute a rich data set of code comments that are classified into the different types of self-admitted technical debt.

III. Approach

The main goal of our study is to identify and quantify the different types of self-admitted technical debt found in source code comments. Figure 1 shows an overview of our approach, and the following subsections detail each step of it.

A. Project Data Extraction

To perform our study, we obtain the source code of five open source projects, namely Apache Ant, Apache Jmeter, ArgoUML, Columba and JFreeChart. We chose the aforementioned projects, since they belong to different application domains, and vary in size (e.g., SLOC), and in the number of contributors.

Table I provides statistics about each one of the projects used in our study. We provide details about the release used, the number of classes, the total source lines of code (SLOC), the total extracted comments and the number of contributors. A source line of code contain at least one valid character, which is not blank spaces or source code comments. In our study, we only use the Java files to calculate the SLOC, and to do so, we use the tool SLOCCount [16].

The number of contributors was extracted from OpenHub, an on-line community and public directory that offers analytics, search services and tools for open source software [17]. It is important to notice that the number of comments shown for each project does not represent the number of commented

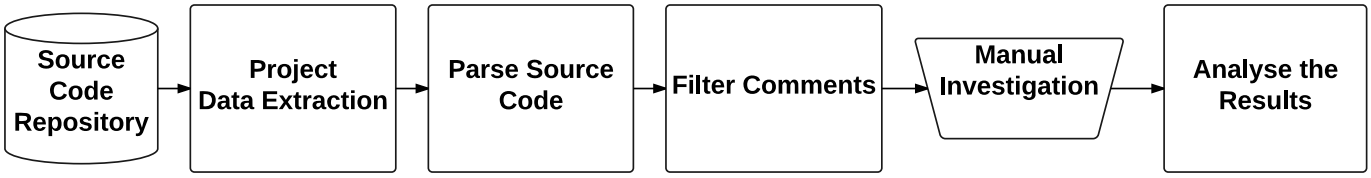

Fig. 1. Approach overview

TABLE I
PROJECT DETAILS

Project	Release	# of classes	SLOC	# of comments	# of contributors
Apache Ant	1.7.0	1,475	115,881	21,587	74
Apache Jmeter	2.10	1,181	81,307	20,084	33
ArgoUML	0.34	2,609	176,839	67,716	87
Columba	1.4	1,711	100,200	33,895	9
JFreeChart	1.0.19	1,065	132,296	23,474	19

lines, but rather the number of individual line, block, and Javadoc comments. In total, we obtained more than 166,756 comments, found in 8,041 Java classes.

B. Parse Source Code

After obtaining the source code of all projects, we extract the comments from their source code. We use JDeodorant [18], an open-source Eclipse plug-in, to parse the source code and extract the code comments. JDeodorant is capable of identify design flaws (i.e., bad smells) in Java projects, and suggest refactoring opportunities to solve them. JDeodrant uses the Eclipse AST framework to create an Abstract Syntax Tree (AST) map of the source code. The AST map contains detailed information about the project such as: the source code comments, its type (i.e., Block, Single-line or Javadoc), the line where each one of these comments begins and finishes. We extract the aforementioned information and store all comments in a relational database to facilitate the processing of the data.

C. Filter Comments

Source code comments can be used for different purposes in a project like giving context, as part of the documentation, to express thoughts, opinions and authorship, and in some cases, to remove source code from the program. Comments are used freely for developers and with few formalities, if any at all. This informal environment allows developers to bring to light opinions, insights and even confessions (e.g., self-admitted technical debt).

As shown in prior work by Potdar and Shihab [3], part of these comments can be identified as self-admitted technical debt, but they are not the majority of cases. With that in mind, we develop and apply 4 filtering heuristics to narrow down the comments eliminating the ones that are less likely to be classified as self-admitted technical debt.

To do so, we developed a Java based tool that reads from the database the data obtained by parsing the source code. Next, it executes the filtering heuristics and stores the result back in the database. The retrieved data contains information like the line number that a class/comment begins/ends and the type, considering the Java syntax, of the comment (i.e., Block, Single-line or Javadoc). With this information we process the filtering heuristics as described next.

We found that license comments are very not likely to contain self-admitted technical debt, and that license comments are commonly added before the declaration of the class. Therefore, we create a heuristic that removes comments that are placed before the class declaration. Since we know the line number that the class was declared we can easily check for comments that are placed before that line and remove them. In order to decrease the chances of removing a self-admitted technical debt comment while executing this filter we calibrated this heuristic to not remove comments containing one of task-reserved words (i.e., “todo”, “fixme”, or “xxx”).

We also notice that some times developers make long comments, using multiple *single-line* comments instead of a Block comment. This characteristic can hinder the understanding of the message. Consider the case that the reader (i.e., human or machine) analyze each one of these comments independently, the message would be incomplete and the meaning lost. To solve that problem, we create a heuristic that searches for consecutive single-line comments and groups them as one. We identify consecutive comments by subtracting the line number of both comments. If the result of the difference is equals a -1 we have a consecutive comment. For example, Single-line comment A is placed in line number 100 and Single-line comment B is placed in line 101. The subtraction of the line numbers will result in -1, therefore the comments are consecutive.

Similarly, is common to find commented source code across the projects, and this can be due to many different reasons. One of the possibilities is that the code is not being used, other is that the code is used for debug purposes only. Based on our analysis, commented source code does not have self-admitted technical debt. Our heuristic remove commented source code using a simple regular expression that captures typical Java

code structures.

Lastly, when analyzing Javadoc comments we found that they rarely mention self-admitted technical debt. For the Javadoc comments that does mention self-admitted technical debt we notice that they usually contains one of the task-reserved words (i.e., "todo", "fixme", or "xxx"). Based on this, our heuristic remove all comments of the type Javadoc unless they contain at least one of the task-reserved words. To do so, we create a simple regular expression that search for the task-reserved words before removing the comment.

The steps mentioned above significantly reduced the number of comments in our dataset and helped us focus on the most applicable and insightful comments. For example, in the Apache Ant project, applying the above steps helped reduce the number of comments from 21,587 to 4,140 comments meaning that 19.17% of the comments were kept for analysis. Table II provides details for each one of the projects.

TABLE II
FILTERING HEURISTICS DETAILS

Project	Total # of comments	# of comments after filtering	% of TD-related comments
Apache Ant	21,587	4,140	19.17 %
Apache Jmeter	20,084	8,163	40.64 %
ArgoUML	67,716	9,788	14.45 %
Columba	33,895	6,569	19.38 %
JFreeChart	23,474	4,436	18.89 %

D. Manual Classification

To classify the comments, we developed a Java based tool that shows one comment at a time and gives a list of possible classifications that can be manually assigned to the comment. The list of possible classifications is based on previous work by Alves *et al.* [15]. After applying the different filtering steps, we successfully classified 33,093 comments. The more than 33 thousand comments were classified into five different types of self-admitted technical debt, i.e., design debt, defect debt, documentation debt, requirement debt and test debt.

The first author who made the classification has more than 8 years of experience working in the industry as a software engineer, during this time he designed, implemented and maintained several programs using, in particular the Java programming language. He developed solid skills in object orientated programming and design patterns. We consider that these qualifications provide the necessary background to conduct the manual classification of the comments.

IV. CASE STUDY RESULTS

The goal of our study is to classify and quantify the different types of self-admitted technical debt. To do so, we divide our study in two parts first, we manually read trough all comments identifying self-admitted technical debt among them. Once identified, the self-admitted technical debt, is classified into different types. Second, we quantify these comments identifying the most common types. Our case study is formalized with the following research question:

RQ: What are the types of self-admitted technical debt? How frequent are the different types of self-admitted technical debt in the studied projects ?

Motivation: As shown in previous work [3], self-admitted technical can be an indicator of non-optimal solutions. However, technical debt is a general term, and there are many different types of technical debt [15]. Although we know that self-admitted technical exists, the different types of self-admitted technical debt are still unknown. For example, are we able to detect documentation debt from code comments? Answering this question is important as different types of debt have different approaches to be solved, and therefore each different type may need a tailored solution. It also helps us understand the opportunities and limitations of using code comments to detect technical debt.

Approach: To identify the different types of debt found in the comments we manually read through source code comments as described in Section III. While examining the comments we classify each comment by the nature of the debt, using the descriptions provided by Alves *et al.* as a guideline.

During the classification we notice that some comments can be classified in more than one type of debt (e.g., a comment reporting a design debt can also be causing an unexpected behavior, which is defect debt). Although this is an ambiguous situation, and may have different interpretations depending of who is reading the comments, we defined that each comment would have just one classification type for the sake of clarity.

To mitigate the chance of misclassifying these comments, we take in consideration the more meaningful type for each comment in a given scenario. To do so, whenever a case like this occurred, we did a more detailed investigation (i.e., by examining the source code and any available documentation). In total we read and classified 33,093 comments from five open source projects. The classification took approximately 95 hours and was performed by the first author of the paper.

Results: We found five different types of self-admitted technical debt. Below, we list the different types of technical debt that we were able to detect and provide example comments to help the reader grasp the different types of self-admitted technical debt comments.

- **Self-admitted design debt:** These comments indicate that there is a problem with the design of the code. They can be comments about misplaced code, lack of abstraction, long methods, poor implementation, workarounds or a temporary solution. Lets consider the following comments:

  "TODO: - This method is too complex, lets break it up" - [from ArgoUml]

  "/ TODO: really should be a separate class */"* - [from ArgoUml]

  These comments are clear examples of what we consider

as self-admitted *design debt*. In the above comments, the developers state what needs to be done in order to improve the current design of the code. Although the above comments are easy to understand, during our study we came across more challenging comments that expressed design problems in an indirect way. For example:

"// I hate this so much even before I start writing it. // Re-initialising a global in a place where no-one will see it just // feels wrong. Oh well, here goes." - [from ArgoUml]

"//quick & dirty, to make nested mapped p-sets work:" - [from Apache Ant]

In the above example comments the authors are certain to be implementing code that does not represent the best solution. Intuitively, we know that kind of implementation will degrade the design of the code and should be avoided.

"// probably not the best choice, but it solves the problem of // relative paths in CLASSPATH" - [from Apache Ant]

"//I can't get my head around this; is encoding treatment needed here?" - [from Apache Ant]

The above comments expressed doubt and uncertainty when implementing the code and were considered as self-admitted design debt as well.

- **Self-admitted defect debt:** In defect debt comments the author states that a part of the code does not have the expected behavior, meaning that there is a defect in the code.

  "// Bug in above method" - [from Apache Jmeter]

  "// WARNING: the OutputStream version of this doesn't work!" - [from ArgoUml]

  As shown in these examples there are defects that are known by the developers, but for some reason is not fixed yet.
- **Self-admitted documentation debt:** In the documentation debt comments the author express that there is no proper documentation supporting that part of the program.

  *"**FIXME** This function needs documentation"* - [from Columba]

  "// TODO Document the reason for this" - [from Apache Jmeter]

  Here, the developers clearly recognize the need to document their code, however, for some reason they do not document it yet.
- **Self-admitted requirement debt:** Requirement debt comments express incompleteness of the method, class or program as observed in the following comments:

  "/TODO no methods yet for getClassname" - [from Apache Ant]

  "//TODO no method for newInstance using a reverse-classloader" - [from Apache Ant]

  *"TODO: The copy function is not yet * completely implemented - so we will * have some exceptions here and there.*/"* - [from ArgoUml]

  The last example shows a comment that could be considered as having more than one type of debt. (i.e., requirement debt and defect debt), but as mentioned in the classification approach, we choose to maintain one type only for each comment. Based on our understanding, the defect debt expressed in the comment would not exist if the requirement debt did not exists. Therefore, the main debt in this comment is a requirement debt (i.e., incomplete implementation of the copy function).
- **Self-admitted test debt:** Test debt comments are the ones that express the need for implementation or improvement of the current tests. As shown in the examples below, test debt comments are very straight forward in their meaning.

  "// TODO - need a lot more tests" - [from Apache Jmeter]

  "//TODO enable some proper tests!!" - [from Apache Jmeter]

After classifying the comments, we notice that not all of the types mentioned in by Alves *et al.* [15] could be found. We argue that some types like people debt or infrastructure debt are less probable to appear in source code comments. Other types such as build debt could not be found because we are examining comments in Java classes only, not taking in consideration build scripts that are usually written in other languages (e.g., Maven and Ant use XML files as build scripts).

> *We find five different types of self-admitted technical debt, i.e., design debt, defect debt, documentation debt, requirement debt and test debt.*

In addition to determining the different type of self-admitted technical debt, we would like to quantify the different types. Doing so will help us understand the strengths and weaknesses of using code comments to detect technical debt. After analyzing the more than 33K comments, we found that only 2,457 comments are self-admitted technical debt comments, representing 7.42% (i.e., $\frac{2457}{33093}$) of all the classified comments. The percentage of self-admitted technical debt found for each project is presented in Table III. ArgoUml is the project with the highest percentage of self-admitted technical debt and Apache Ant has the lowest percentage, amounting to 16.8% and 3.2% respectively.

Figure 2 shows the percentage of each type of self-admitted technical debt across the projects. Since each project has a different number of comments we normalized the data, presenting the percentages of the different types rather than the raw numbers. For example, if a project has 100 self-admitted technical debt comments and 10 where design debt type, we say that the project has 10% of self-admitted design technical debt.

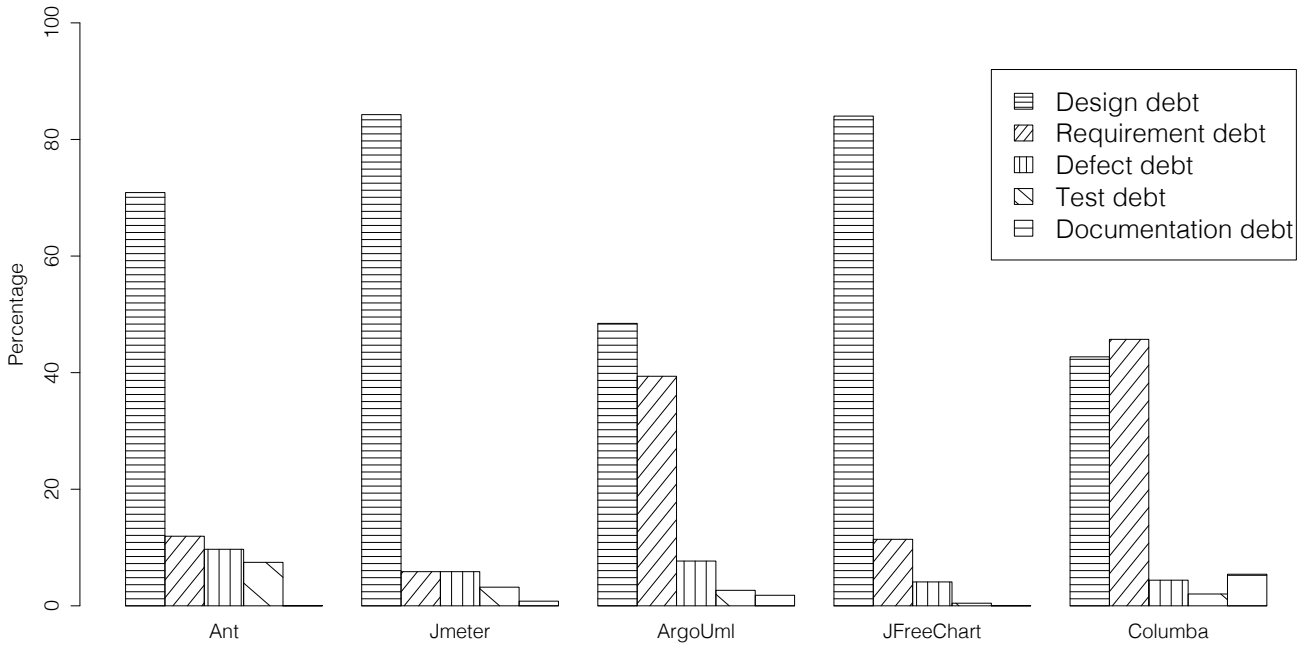

Fig. 2. Self-admitted technical debt types distribution

TABLE III
SELF-ADMITTED TECHNICAL DEBT PER PROJECT

Project	# of analyzed comments	# of self-admitted TD comments	% of self-admitted TD per project
Apache Ant	4,140	134	3.2
Apache Jmeter	8,163	375	4.6
ArgoUML	9,788	1,653	16.8
Columba	6,569	295	4.4
JFreeChart	4,433	219	4.9

Analyzing the Figure 2 we find that self-admitted design debt is the most common in 4 out of 5 projects. Self-admitted design technical debt values ranged from 42%, in Columba project with the lowest percentage, to 84% in Jmeter and JFreeChart, projects with the highest percentage. The second most frequent type is self-admitted requirement debt with values between 5% and 45%, followed by self-admitted defect technical debt making up between 4% to 9% of the comments. Self-admitted test technical debt ranged from 0% to 7% whereas self-admitted documentation debt had only 0% to 5% of the comments.

We notice that Columba and ArgoUml have the highest occurrences of self-admitted requirement debt. Columba is a email client application written in Java, which has 9 contributors [17], and a considerable number of classes 1,711. It is reasonable to think that developers have limited time to develop features. Therefore, leaving comments of features that need to be implemented in the future (i.e., requirement debt) is more likely.

ArgoUml has a high number of contributors i.e., 87 and yet has a hight number of self-admitted requirement debt. Analyzing the comments we notice that there occurrences about the need of support for internationalization and other comments express the need to implement code to make features compatible with newer versions of the UML language.

Based on that we argue that coupling with external changes that are inherent of the application domain and the adoption of the tool from users all over the world [17] had increased the number of self-admitted requirement debt.

We find that the majority of the self-admitted technical debt comments are design debt, which ranged from 42% to 84% across the projects. The second most frequent type was requirement debt that ranged from 5% to 45%. The remaining types have low frequency if considered that they represented less than 10% of the occurrences

V. THREATS TO VALIDITY

Internal validity consider the relationship between theory and observation, in case the measured variables do not measure the actual factors. To classify the source code comments we heavily depended on manual process due the fact that comments are written in natural language and therefore difficult to analyze by a machine. Like any human activity, our manual classification is subject to personal bias and subjectivity. To reduce this bias, in the future, we will ask to other researchers of our lab to classify the dataset as well, verifying and discussing possible

divergences of opinion. This is important as changes in this dataset may impact our findings.

When performing our study, we used well-commented Java projects. Since our technique heavily depends on code comments, our results may be impacted by the quantity and quality of comments in a software project. To alleviate the threat, we examined multiple projects. Moreover, there is a risk of removing self-admitted technical debt comments while filtering license comments. To mitigate this risk we do not remove comments that contain one of task-reserved words (i.e., "todo", "fixme", or "xxx").

External validity consider the generalization of our findings. All of our findings were derived from comments in open source projects. To minimize external validity, we chose open source projects from different domains. That said, our results may not generalize to other open source or commercial projects. In particular, our results may not generalize to projects that have a low number or no comments. Other than that, we only analyze projects written in Java, therefore the results obtained may not generalize to projects written in other languages.

VI. Conclusion and Future work

The term technical debt is being used for practitioners and researchers in the software engineer community to express shortcuts and workarounds employed in software projects. These shortcuts will most often impact the maintainability of the project hindering the development if not addressed properly. Our work explore specifically self-admitted technical debt, that is the technical debt deliberately introduced by the developers and reported through source code comments.

In our study we analyzed the comments of 5 open source projects which are Apache Ant, Apache Jmeter, ArgoUml , Columba and JFreeChart. These projects are considered well commented and they belong to different application domains. We used them to understand the characteristics of self-admitted technical debt types creating a rich dataset with more than 33,093 classified comments.

We find that self-admitted technical debt can be classified into five types: design debt, defect debt, documentation debt, requirement debt and test debt. We also provide concrete examples of each one of the mentioned types and the rationale to classify them as it was. Moreover, we find that the majority of the self-admitted technical debt comments are design debt. Design debt ranged from 42% to 84% across the projects. The second most frequent type was requirement debt ranging from 5% to 45%. Based on this result, we can say that the self-admitted technical debt types that developers admit to the most are related with the design of the project, potentially indicating that developers feel the need to admit and be forthcoming about such debt. Examining the reasons for these types of debt is an interesting future direction that we plan to pursue.

Other contribution of our study is that we make publicly available the resulting dataset of our classification. We hope that this will encourage future research in the area of self-admitted technical debt as, to the best of our knowledge, this is the first dataset of this kind. We also think that the information provided by this dataset can be a cornerstone for more advanced techniques as natural language processing.

In a future work we plan to improve the current classification adding more projects to it. With a richer dataset we expect that more patterns and characteristics of the self-admitted technical types will be retrieved. We also plan to use this database to mine unique sequential patterns, an advanced technique of natural language processing, which may lead to more automated ways to identify self-admitted technical debt.

References

[1] W. Cunningham, "The wycash portfolio management system," in *Addendum to the Proceedings on Object-oriented Programming Systems, Languages, and Applications (Addendum)*, ser. OOPSLA '92. New York, NY, USA: ACM, 1992, pp. 29–30. [Online]. Available: http://doi.acm.org/10.1145/157709.157715

[2] D. Falessi, P. Kruchten, R. L. Nord, and I. Ozkaya, "Technical debt at the crossroads of research and practice: report on the fifth international workshop on managing technical debt," *ACM SIGSOFT Software Engineering Notes*, vol. 39, no. 2, pp. 31–33, 2014. [Online]. Available: http://doi.acm.org/10.1145/2579281.2579311

[3] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 91–100.

[4] 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*, 2013, pp. 42–47.

[5] N. Zazworka, M. A. Shaw, F. Shull, and C. Seaman, "Investigating the impact of design debt on software quality," in *Proceedings of the Second Workshop on Managing Technical Debt*, 2011, pp. 17–23.

[6] F. Fontana, V. Ferme, and S. Spinelli, "Investigating the impact of code smells debt on quality code evaluation," in *Proceedings of the Third International Workshop on Managing Technical Debt*, 2012, pp. 15–22.

[7] B. Fluri, M. Wursch, and H. 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*, 2007, pp. 70–79.

[8] S. H. Tan, D. Marinov, L. Tan, and G. Leavens, "@tcomment: Testing javadoc comments to detect comment-code inconsistencies," in *Proceedings of the IEEE Fifth International Conference on Software Testing, Verification and Validation*, 2012, pp. 260–269.

[9] H. Malik, I. Chowdhury, H.-M. Tsou, Z. M. Jiang, and A. Hassan, "Understanding the rationale for updating a function comment," in *Proceedings of the IEEE International Conference on Software Maintenance*, 2008, pp. 167–176.

[10] M. Storey, J. Ryall, R. Bull, D. Myers, and J. Singer, "Todo or to bug," in *Proceedings of the ACM/IEEE 30th International Conference on Software Engineering*, 2008, pp. 251–260.

[11] C. Seaman and Y. Guo, "Measuring and monitoring technical debt," in *Advances in Computers*, M. V. Zelkowitz, Ed. Elsevier, 2011, vol. 82, pp. 25–46.

[12] 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*, vol. 38, no. 5, pp. 51–54, Aug. 2013.

[13] 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*, 2010, pp. 47–52.

[14] R. Spinola, N. Zazworka, A. Vetro, C. Seaman, and F. Shull, "Investigating technical debt folklore: Shedding some light on technical debt opinion," in *Managing Technical Debt (MTD), 2013 4th International Workshop on*, May 2013, pp. 1–7.

[15] N. Alves, L. Ribeiro, V. Caires, T. Mendes, and R. Spinola, "Towards an ontology of terms on technical debt," in *Managing Technical Debt (MTD), 2014 Sixth International Workshop on*, 2014, pp. 1–7.

[16] D. A. Wheeler, "Sloc count users guide," 2004.

[17] "OpenHub homepage," https://www.openhub.net/, accessed: 2014-12-12.
[18] N. Tsantalis, T. Chaikalis, and A. Chatzigeorgiou, "Jdeodorant: Identification and removal of type-checking bad smells," in *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 329–331.