Rework Effort Estimation of Self-admitted Technical Debt

Solomon Mensah[1], Jacky Keung[1], Michael Franklin Bosu[2] and Kwabena Ebo Bennin[1]
[1]Department of Computer Science, City University of Hong Kong, Hong Kong, China
{smensah2-c, kebennin2-c}@my.cityu.edu.hk, Jacky.Keung@cityu.edu.hk
[2]Centre for Business, Information Technology and Enterprise
Wintec, Hamilton, New Zealand
michael.bosu@wintec.ac.nz

Abstract—**Programmers sometimes leave incomplete, temporary workarounds and buggy codes that require rework. This phenomenon in software development is referred to as Self-admitted Technical Debt (SATD). The challenge therefore is for software engineering researchers and practitioners to resolve the SATD problem to improve the software quality. We performed an exploratory study using a text mining approach to extract SATD from developers' source code comments and implement an effort metric to compute the rework effort that might be needed to resolve the SATD problem. The result of this study confirms the result of a prior study that found design debt to be the most predominant class of SATD. Results from this study also indicate that a significant amount of rework effort of between 13 and 32 commented LOC on average per SATD prone source file is required to resolve the SATD challenge across all the four projects considered. The text mining approach incorporated into the rework effort metric will speed up the extraction and analysis of SATD that are generated during software projects. It will also aid in managerial decisions of whether to handle SATD as part of on-going project development or defer it to the maintenance phase.**

Keywords—Self-admitted Technical Debt; Rework Effort; Text Mining; Source code comments; Source code analysis

I. INTRODUCTION

The increasing pressure to deliver fast software products to customers sometimes forces project managers to impose unrealistic deadlines on their developers. As a result, these developers intentionally commit incomplete code, buggy code and temporary fixes in order to meet the expectation of their customers. This practice could produce errors which might require rework. These intentional or self-admitted errors are assumed as mistakes by the software development team. Potdar and Shihab [1] describe this phenomenon of weak software development process resulting in series of long-term overheads in the maintenance phase as Self-admitted Technical Debt (SATD). The debt metaphor is gradually becoming a research focus [1][3][5] with studies aimed at finding solutions for combating or minimizing the developers' coding errors and shortcuts of producing less quality applications [6].

Harrington's concept of "cost of poor quality" [7] in relation to technical debt basically refers to the cost involved in resolving defective products. According to Chatzigeorgiou et al. [8], the concept of "cost of poor quality" does not only deal with the cost for rectifying the gap between optimum and actual products but also involves the effort required to resolve defects in delivered products.

The challenging question that arises among project managers prior to release of software product is "*Should we meet our short-term business objective and release the product as soon as possible or we should take our time and fix the code before release?*" From either point of view, a loss or debt in relation to software quality can be incurred. It is worth noting that not all SATD can realistically be repaid. In this study, the effort involved in resolving these debts is described as *Rework Effort*. Rework effort from the point of view of Bhardwaj and Rana [11] plays a significant role in software testing and leads to additional cost in software development. For a released product to be more robust and long-term effective, there is the need to consider the amount of rework effort that is needed to fix all identified SATD in the software project.

To study the issue of this debt metaphor, we extracted source code comments from four large open-source software projects and performed an exploratory study analysis on the corpus of code comments with the intention of estimating the rework effort necessary to fix the SATD tasks. Based on a vocabulary of SATD indicators manually identified by Potdar and Shihab [1], we developed an automated text mining approach to assist in the extraction and estimation of the rework effort for SATD tasks. We classify the SATD tasks into five classes based on the classification scheme by Maldonado and Shihab [3] using the algorithm in Section C. The contribution of this work is twofold: to the best of our knowledge this is the first study to use text mining in identifying SATD from source code comments and to estimate rework effort of SATD.

The remaining sections of the paper are organized as follows. Section II highlights the methodological procedure employed. Section III addresses the results from the empirical analysis of the study. Finally, Section IV presents the threats to validity and Section V gives a summary of the study based on conclusions and future directions of the study.

II. METHODOLOGY

The exploratory analysis for this study was performed using the MATLAB toolkit (version R2014b) and the R Software (version 3.2.2). These toolkits enabled in the setting up of the text mining algorithm by constructing regular expressions for the source code analysis and searching for patterns for SATD from the open-source projects.

A. *Datasets*

For the purpose of this study, we chose four well-commented open-source projects made available at openhub.net. These datasets were first extracted by Potdar and Shihab [1] for a manual exploratory study of SATD. The four projects are ArgoUML, Chromium OS, Apache HTTP Server and Eclipse Platform project. The description of the open-source projects is presented in Table I. In each project, the following metrics were extracted - the total number of Lines of Code (LOC), lines of source code comments, contributors or developers and the dates of software release.

TABLE I. DESCRIPTION OF OPEN-SOURCE PROJECTS

	Open-Source Projects			
Metric	**AgroUML**	**Chromium**	**Eclipse**	**Apache**
LOC	122,575	107,706	659,231	192,333
Comment lines	115,713	37,889	437,640	54,295
Release Date	Dec, 2011	Nov, 2009	Jun, 2013	Jul, 2013
Developers	53	1,784	221	145
Version	0.34	30	4.3	2.4.6

B. *Data Preprocessing Methods*

Preprocessing is an important phase in text mining and text classification. For an efficient regular expression matching, we preprocessed the extracted open-source code comments based on data cleaning, stopword filtering, and term weighting. In the dataset cleaning process, we used the text mining approach to remove punctuation marks in the form of ~!@,.-#$%^*][|\ from the corpus of code comments. Again, we filtered out noise in the form of blank lines and white spaces within strings from each project. Stopwords occurring frequently (such as *and*, *this*, *the*, *or*, *of*, *am*, *it*, *on*, *at*) were removed because they contributed less in the text mining and classification process. These words were searched and removed following an approach by Fabrizio [10]. We assigned term weights to the various SATD code comments in all cases of the project datasets to know the frequency at which the SATD indicators occurred in the source code comments. The assignment of term weights was done based on term frequency-inverse document frequency (*tfidf*) [4] which is a well-known ranking function in text mining and information retrieval. The *tfidf* function is composed of the product of the term frequency (*tf*) and the inverse document frequency (*idf*). We define these two terms in (1) and (2) with respect to each project dataset.

$$tf(t,d) = \frac{f_{t,d}}{m_d} \quad \forall d \in D \tag{1}$$

$$idf(t,D) = \log_e \frac{D}{N_t} \tag{2}$$

$$tfidf(t,d,D) = tf(t,d) \times idf(t,D) \tag{3}$$

where $f_{t,d}$ = frequency of term (t) in an SATD comment (d)
m_d = number of terms in a given SATD comment
D = total number of SATD comments per source file
N_t = number of SATD comments with a given term (t)

C. *Proposed Text Mining Technique*

We proposed a text mining technique (Algorithm 1) for mining SATD tasks using source code comments. This technique plays a significant role in transforming source code comments into numeric counts based on the assignment of term weights for easy modeling and rework effort estimation. The text mining technique for commented source code is divided into 5 phases as follows:

Phase I: Preprocessing phase of the project datasets
Phase II: Extraction of code comments containing SATD
Phase III: Categorization of SATD classes
Phase IV: Computation of term weights for SATD tasks
Phase V: Computation of Rework Effort for SATD tasks

Provision of some notations of the various variable names used is made available. The algorithm constructed with regular expressions is supplied with the contributor/developer details and their respective comments made. Prior to Phase I, we employed the *textscan* function to read the separated strings in each of the code comments into separate vectors for each system studied. This function also contributed in reading commented strings with whitespaces.

In Phase I, punctuation and special characters such as {“ ”:\;!/.@[]-?#%^()’ ’} were eliminated from each of the source code comment and contributor using the *punct[ ]* function and result assigned to the variable *P* (line 1). Stop words such as *is*, *are*, *of*, *the*, *that*, *with*, *a*, *so*, *to*, *by*, *but*, *if*, *it*, *and*, *in*, *what*, *how* and other related words were removed in line 2 and the remaining result assigned to *SW* variable.

In Phase II, SATD comments void of stop words were extracted using an implemented *extract_satd* function containing the array of SATD indicators [1] in the first *for loop* from lines 3 to 5.

In Phase III, we made use of a dictionary of indicators, *StD_type* for the various types of SATD tasks as studied by Maldonado and Shihab [3]. Thus, with the help of this dictionary, we can search and extract the various types of SATD tasks in line 7.

With the help of the *tfidf* for each case, statistical analysis was made on the transformed dataset for statistical inferences. In Phase IV, we made use of *tfidf* [4] in the second for loop statement from lines 9 to 13 to compute the term weights for the SATD list. In line 10, the total number of terms per each comment within each corpus was computed and the term frequency computed in line 11 as the ratio of the number of searched and targeted *t* terms to the end result in line 10. We computed the inverse document frequency in line 12 ignoring case sensitiveness of terms in the *grepl*[1] function. The *grepl* function returns a logical vector containing searched SATD comments. The *tfidf* values were computed in line 13 for each SATD code comment. In Phase V, the rework effort (*RW*) is computed in step 16 and further explained in equation (4).

[1] *grepl is a function in the CRAN library of R which returns a particular string when found in the search space.*

Algorithm 1 Source Code Comment Text Mining

```
Notations:
    P: remove punctuations' function
    SW: remove stop words' function
    Q: total number of commented tasks per project
    D: total number of SATD commented tasks per project
    SATD: List of SATD comments
    class: Class of SATD indicators
    tfidf: term frequency inverse document frequency
Input:
    DCS: Dataset of contributors and source code comments
    StW[ ]: array of stopwords
    punct[ ]: array of punctuation characters
    StD: array of SATD indicators
    StD_type: array of types of SATD indicators
    RsF: rank source files
Output:
    RW: Rework Effort for SATD tasks
Procedure
    // Remove Punctuation Characters
1:  P ← remove_punct("punct[]", DCS)
    // Remove Stop Words
2:  SW ← remove_stop.words(P, StW[])
    //Extract SATD comments from corpus
3:    for i, i=1,...,Q do
4:        SATD[i] ← extract_satd(P[i], StD)
5:    end for
    // Categorization of SATD Tasks
6:    for l, l=1,...,D do
7:        class[l] ← categorize(StD_type[l])
8:    end for
    // Compute term weights for SATD list using tfidf
9:    for j, j=1,...,D do
    //Computing number of terms(t) per each SATD comments
10:       tf_tot[j] ← compute(SATD[j], length)
11:       tf[j] ← count(t terms) / tf_tot[j]
12:       idf[j]← log(D / sum(grepl(SATD[j], ignore.case)))
13:       tfidf[j] ← tf[j] * idf[j]
14:       k ← cos(RsF, StD)
15:       $S_k$ ← count(StD, file[k])
          //Computation of Rework Effort
16:     RW ← compute(LOC[j]/$S_k$)
17:   end for
18:   Output RW
```

D. Rework Effort Estimation Metric for SATD

In the quest of investigating the extent of rework effort in relation to resolving commented LOC prone to SATD, we formulated a rework effort metric based on a study by Zhao et al. [2]. The rework effort (*RW*) metric is defined as follows:

$$RW = \frac{\sum_{j=1}^{n}\sum_{i=1}^{k} LOC(F_{ij})}{S_k} \quad (4)$$

where $LOC(F_{ij})$ denotes the commented LOC of the ith source file in the ranked list for the jth SATD indicator. S_k is the number of SATD indicators contained in the *k* ranked source files (step 15). *n* is the total number of SATD indicators. Thus, given any software project containing *n* commented LOC in a number of source files, we first compute the term weights of the source files, followed by a ranking process [2] and use the cosine similarity to obtain the *k* ranked source files. The cosine similarity finds the close relation between the source files and SATD indicators [1] with the intention of obtaining *k* files prone to SATD (step 14). The *k* SATD prone files were obtained based on a cosine similarity threshold of at least 0.7. In relation to each k^{th} file, we extract the commented LOC that contains SATD. This is done repeatedly until all the commented LOC tasks are obtained from the *n* source files as the numerator in (4). *RW* is computed as the ratio of the numerator ($LOC(F_{ij})$) and denominator (S_k). We present a sample of the code comments prone to SATD below.

Examples of SATD comments

* Don't wait around; just abandon it *
* Leave it for next release *
* Do nothing and bail out *
* Strictly speaking, this is a design error *
* DESIGN ERROR: a mix of repositories *
* TODO: this isn't quite right but is ok for now *

This list of SATD indicators [1] formed the vocabulary of words which was used in the proposed text mining approach for the rework effort estimation. With respect to previous study [3], the SATD commented tasks were categorized into five classes – requirement debt, design debt, testing debt, defect debt and documentation debt. The explanation with examples of the classes of SATD are elaborated in [3].

We evaluated the classification performance of the proposed text mining approach by averaging the precision and recall values across the 4 open-source projects.

III. Results

A. RQ1: What is the dominant class of self-admitted technical debt?

Question *RQ1* is similar to the one posed in [3]. Because we used different datasets from those used in [3], we decided to test the postulation that design debt is the predominant class of SATD in each of the open-source projects. The distribution of this class of debt was irrespective of the size of the project. For example, Apache project with 452 SATD comments had design debt of 62.1%, Eclipse with 167 SATD comments had design debt of 56.5%. Similarly, the design debts for AgroUML (512 SATD comments) and Chromium (975 SATD comments) were 56.5% and 67.5% respectively. Clearly, all design debts are more than 50% of SATD comments in each project. This result confirms a similar result by Maldonado and Shihab [3] that found that design debt contributes between 42% and 84% of all identified SATD in different systems.

Precision (*P*) and Recall (*R*) values of confusion matrices created from the text mining approach for the classification were as follows: requirement debt (*P=0.84, R=0.77*), design debt (*P=0.85, R=0.84*), testing debt (*P=0.87, R=0.92*), defect debt (*P=0.76, R=0.82*) and documentation debt (*P=0.81, R=0.79*).

B. *RQ2: What is the extent of rework effort required to resolve SATD in open-source projects?*

Table 2 indicates the estimated rework effort (measured in average commented LOC per SATD prone source file of each system) for the maintenance team to resolve these SATD within the source files of the respective systems studied. It should be noted that *Req't* and *Docu* in Table 2 denote Requirement and Document debts respectively. From the perspective of considering all the five classes of debts, it was realized that design debt required substantial rework effort as elaborated in Table 2. Thus, the rework effort for resolving design debt in AgroUML is 7.9, Chromium is 17.1, Eclipse is 11.8 and lastly, Apache is 12.6 commented LOC on average per SATD prone source file. Similarly, test and defect debts were also of key interest in this study which needed rework apart from design debts. These two debts even though known by the development team that it will lead to long-term bugs upon release were left unfixed. This we believe will be due to the time-to-market constraint as mentioned by Fernández-Sánchez et al. [9].

Based on results from Table 2, there is no unique pattern in relation to the SATD rework effort and the size of the open-source projects. A typical example is seen in Eclipse and Apache. Even though Eclipse has 437,640 commented LOC much larger than that of Apache with 54,295 (Table 1), the amount of SATD rework effort for Eclipse is 11.8 as compared to 12.6 in Apache (Table 2). It can be seen that the rework effort estimation of about 13-32 commented LOC on average per SATD prone source file across the selected projects could affect the quality of the software product.

TABLE 2: REWORK EFFORT FOR RESOLVING SATD

SATD Class	**Rework Effort for Projects**			
	Agro	**Chromium**	**Eclipse**	**Apache**
Req't	0.7	2	4.4	3.9
Design	7.9	17.1	11.8	12.6
Testing	3.1	4.6	5.1	7.3
Defect	1.6	5.3	3.1	5.6
Docu.	0	0.4	0.3	2.2
Total	13.3	29.4	24.7	31.6

IV. THREATS TO VALIDITY

The first threat to validity in this study is the use of well-commented open-source project datasets. This constraint might not be a representative sample of the total population of open-source projects since not all projects are well-commented. Thus, the findings of this study cannot confidently be generalized. The selected projects used are popular and large in size. Therefore, the examination of all the developers' comments from the projects with the intention of resolving the self-admitted technical debt (SATD) problem can form a good foundation for researchers to conduct more in-depth studies in this field. Secondly, the list of SATD indicators used from previous study might not be a generalized representation of all SATD in the software development and maintenance environment. Since this study focused on source code comment analysis, we were constraint of gathering more information especially from industry to validate the results obtained.

V. CONCLUSION

In this study, we performed an exploratory analysis with a proposed text mining approach on source code comments of four open-source projects. With the help of transforming the source code comments into term weights, we were able to estimate the rework effort for fixing these debts. This study addressed two main research questions:

RQ1: What is the dominant class of self-admitted technical debt?

Results from the study indicate that out of all the five classes of SATD, design debts (56.5% - 67.5%) is the predominant class of SATD for all the four systems.

RQ2: What is the extent of rework effort required to resolve SATD in open-source projects?

The result of this study indicate that rework effort of between 13 and 32 commented LOC on average per SATD prone source file will have be addressed in order to fix the SATD. In order to improve the long term quality of the software, it is essential that developers are encouraged to avoid SATD.

The proposed approach is a novel technique which can assist in the estimation of rework effort needed to fix SATD tasks that demands rework.

In going forward, we intend to validate our approach based on industrial case studies and different versions of open-source datasets to facilitate result generalization.

REFERENCES

[1] A. Potdar, and E. Shihab. "An exploratory study on self-admitted technical debt." *2014 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 2014.

[2] F. Zhao, Y. Tang, Y. Yang, H. Lu, Y. Zhou, and B. Xu. "Is Learning-to-Rank Cost-Effective in Recommending Relevant Files for Bug Localization?." *Software Quality, Reliability and Security (QRS), 2015 IEEE International Conference on*. IEEE, 2015.

[3] E. S. Maldonado, and E. Shihab. "Detecting and quantifying different types of self-admitted technical Debt." *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE, 2015.

[4] C. D. Manning, P. Raghavan and H. Schütze. *Introduction to information retrieval*. Vol. 1. No. 1. Cambridge: Cambridge university press, 2008.

[5] W. Sultan, E. Shihab, and L. Guerrouj. "Examining the Impact of Self-admitted Technical Debt on Software Quality." *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*. IEEE, 2015.

[6] Y. Padioleau, T. Lin, and Z. Yuanyuan. "Listening to programmers—Taxonomies and characteristics of comments in operating system code." *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on Software Engineering*. IEEE, 2009.

[7] H. J. Harrington, "Poor-Quality Cost: Implementing, Understanding, and Using the Cost of Poor Quality (Quality and Reliability)." (1987).

[8] A. Chatzigeorgiou, et al. "Estimating the breaking point for technical debt." *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE, 2015.

[9] C. Fernández-Sánchez, J. Garbajosa, and A. Yagüe. "A framework to aid in decision making for technical debt management." *Managing Technical Debt (MTD), 2015 IEEE 7th International Workshop on*. IEEE, 2015.

[10] S. Fabrizio. "Machine learning in automated text categorization."*ACM computing surveys (CSUR)* 34.1 (2002): 1-47.

[11] M. Bhardwaj, and A. Rana. "Impact of Size and Productivity on Testing and Rework Efforts for Web-based Development Projects." *ACM SIGSOFT Software Engineering Notes* 40.2 (2015): 1-4.