The Journal of Systems and Software 135 (2018) 37–54

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

On the value of a prioritization scheme for resolving Self-admitted technical debt

Solomon Mensah[a,*], Jacky Keung[a], Jeffery Svajlenko[b], Kwabena Ebo Bennin[a], Qing Mi[a]

[a] Department of Computer Science, City University of Hong Kong, Hong Kong, China
[b] Department of Computer Science, University of Saskatchewan, Saskatoon, Canada

ARTICLE INFO

Article history:
Received 12 February 2017
Revised 16 August 2017
Accepted 25 September 2017
Available online 28 September 2017

Keywords:
Self-admitted technical debt
Prioritization scheme
Textual indicators
Source code comment
Open source projects

ABSTRACT

Programmers tend to leave incomplete, temporary workarounds and buggy codes that require rework in software development and such pitfall is referred to as Self-admitted Technical Debt (SATD). Previous studies have shown that SATD negatively affects software project and incurs high maintenance overheads. In this study, we introduce a prioritization scheme comprising mainly of identification, examination and rework effort estimation of prioritized tasks in order to make a final decision prior to software release. Using the proposed prioritization scheme, we perform an exploratory analysis on four open source projects to investigate how SATD can be minimized. Four prominent causes of SATD are identified, namely code smells (23.2%), complicated and complex tasks (22.0%), inadequate code testing (21.2%) and unexpected code performance (17.4%). Results show that, among all the types of SATD, design debts on average are highly prone to software bugs across the four projects analysed. Our findings show that a rework effort of approximately 10 to 25 commented LOC per SATD source file is needed to address the highly prioritized SATD (*vital few*) tasks. The proposed prioritization scheme is a novel technique that will aid in decision making prior to software release in an attempt to minimize high maintenance overheads.

1. Introduction

The concept of technical debt, first introduced by Cunningham (1993), refers to the debt incurred through the speeding up of software project development which results in a number of deficiencies ending up in high maintenance overheads. These deficient and imperfect traits in the software development prior to project release will have to be paid in the near future as an uncontrolled maintenance cost. The technical debt metaphor, a major issue in software development and maintenance, negatively affects software quality and has recently been a focus for several research studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Ramasubbu, 2013, Fernandez-Sanchez et al., 2015, Kruchten et al., 2012, Maldonado and Shihab, 2015, Wehaibi et al., 2016, Mensah et al., 2016).

With the increasing pressure of expediting software products for users, project managers obliged to meet stipulated deadlines and short-term business benefits, tend to impose pressure on their programmers. As a result, these programmers commit incomplete codes, buggy codes and temporary fixes that produce errors and require rework so as to meet the need of customers who demand quality and robust applications. These errors are assumed as intentional mistakes by the software development team. Potdar and Shihab (2014) describe this phenomenon of intentional weak software development process resulting in series of long-term overheads in the maintenance phase as Self-admitted Technical Debt (SATD). Thus, SATD refers to the incomplete or temporary fixes which are intentionally committed by developers and admitted as mistakes during software development. The SATD metaphor is gradually becoming a research focus (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015, Wehaibi et al., 2016) whereby researchers aim at finding solutions for combating or minimizing the developers' misconducts and shortcuts of producing less quality applications. This misconduct in development is sometimes based on decisions that prioritize functionality over quality (Fernandez-Sanchez et al., 2015) and goes a long way to negatively affect software maintenance (Zazworka et al., 2011).

A plethora of prior studies Zimmermann et al. (2008) and Fluri et al. (2007) conducted in recent years focused on software quality issues in relation to software bugs introduced by developers and assumed such bugs as mistakes. However, very little is known about the proportion of different kinds of SATD in a software project. Knowledge of the proportion could help software practitioners prioritize their scarce testing resources. A potential way of determining the proportions is by examining the source

* Corresponding author.
E-mail addresses: smensah2-c@my.cityu.edu.hk (S. Mensah), Jacky.Keung@cityu.edu.hk (J. Keung), jeff.svajlenko@usask.ca (J. Svajlenko), kebennin2-c@my.cityu.edu.hk (K.E. Bennin), Qing.Mi@my.cityu.edu.hk (Q. Mi).

https://doi.org/10.1016/j.jss.2017.09.026
0164-1212/

code comments of the software project. Software comments (auxiliary segments which are not compiled) play a key role in understanding the source codes in relation to software development and maintenance. Inspired by previous works by Potdar and Shihab (2014) and Fluri et al. (2007), we perform an exploratory study on source code comments of open source projects whereby SATD tasks are prioritized to know the extent of its negative effects on software quality. According to Devroey et al. (2013), prioritization can be used to sort items and optimize coverage criteria or assign weights to features. Prioritization of technical debt items is an important activity in the software engineering domain since it balances short-term product release and long-term effects associated with the software release (Martini et al., 2014). According to Li et al. (2015), there is the need for more studies to be conducted on how to prioritize technical debt task list in order to maximize profit.

In the attempt to deal with the prioritization of the SATD metaphor, we introduce a 6-step prioritization scheme to provide a framework for minimizing SATD based on identified textual indicators by Potdar and Shihab (2014). We construct a text mining algorithm to mine indicative source code comments of SATD from four large open source software projects. Thus, we make use of extracted source code comments and empirically investigate the causes of SATD and estimate the rework effort involved in addressing them. We empirically validate the proposed prioritization approach on four open source projects using Recall and Precision accuracy measures. We further perform statistical tests using the Welch's *t-test* and Cliff's δ effect size as recommended by Kitchenham et al. (2016) to confirm the statistical significance of our results.

Results from this study show that about 30–38% of the identified SATD tasks were major tasks which on average developers face difficulty in addressing prior to software release. This resulted in a rework effort estimation of approximately 10–25 commented LOC per SATD source file needed to address highly prioritized SATD or *vital few* tasks. Finally, seven distinct causes of SATD were found with code smells, complex tasks, inadequate code testing and unexpected code performance being the most dominant causes. The prioritization scheme can form a decision-making baseline prior to software release to assist software engineers in minimizing SATD and its related impacts during software development.

Main contributions of this study include:

- a prioritization scheme for addressing SATD tasks.
- a text mining algorithm for SATD detection and prioritization.
- a rework effort estimation of prioritized SATD tasks.

The remaining sections of the paper are organized as follows. Section 2 presents reviews of related work. Section 3 presents details of the 6-step prioritization scheme as well as the methodological procedure employed in conducting the study. Section 4 discusses the results from the empirical analysis of the study. Section 5 presents the threats to validity and Section 6 gives the conclusions and future direction of the study.

2. Related work

Several works have been done in the field of technical debt in general (Ramasubbu, 2013, Li et al., 2015, Farias et al., 2015, Kruchten et al., 2012, Martini et al., 2014). Farias et al. (2015) performed an exploratory analysis on two large open source projects to identify technical debts through source code comment analysis. They proposed a model, namely CVM-TD which provides a vocabulary that can be used to detect technical debts. Results from their study indicate that developers use dimensions considered by CVM-TD model to assist them in writing source code comments during development. They concluded from their study that technical debts can be identified from source code comment analysis. Ramasubbu (2013) developed and validated the theory of accumulation of technical debt that measures the impact of tradeoffs between customer satisfaction and software quality. Their study was performed using a commercial software product at the various stages of the product's lifecycle. Their proposed theory provides relevant cost estimation and benefits of technical debt for the adoption of a commercial software product.

In relation to our study, we focused on technical debts which are self-admitted by the development team. The concept of SATD was introduced by Potdar and Shihab (2014) who performed an exploratory study on four large open source projects to study the amount, reason and likelihood of removing SATD after its introduction during software development. In their study, they manually examined the open source code comments and found 62 textual indicators for identifying SATD. Out of the 2.4–31% of source files that contained SATD, most of them were introduced by more experienced developers as compared to less experienced developers. They also found that time and code complexities do not correlate with the amount of SATD. Lastly, they found that even though these SATD are meant to be eliminated after its introduction, close to 30% were not addressed or removed from the software projects analysed. A follow-up study by Maldonado and Shihab (2015) identified and quantified SATD from at least 33,000 source code comments into five main types, namely requirement debt, design debt, test debt, defect debt and documentation debt. They found that design debts were the most common type of debt forming about 42–84% of the detected SATD comments. Another study by Wehaibi et al. (2016) examined the impact of SATD on software quality. They found that changes made on SATD tasks induced less defects in future as compared to non-SATD tasks and that these changes are more complex to be made. Bavota and Russo (2016) recently conducted a *differentiated replication* (Wohlin et al., 2012) study of Potdar and Shihab's work to investigate the diffusion rate of SATD in software development. In their study, they (Bavota and Russo, 2016) mined over 2 billion source code comments of 159 Java open source projects and identified an average of 51 SATD instances per each project. In their manual categorization of SATD inspired by the Grounded theory principles (Corbin and Strauss, 1990), they found that 30% of the SATD instances was code debt, 20% represented requirement and defect debts, and 13% represented design debt. They also found that the number of SATD instances increases with the change history, and in most cases (63%) the same developers who introduced the debt were the same fixing the debt. Another recent study by Mensah et al. (2016) estimated the rework effort needed to address the SATD tasks in open source projects. Result from the study shows that, a rework effort of approximately 13–32 commented LOC is required to address the SATD tasks in each source file of the software projects analysed.

Even though this study makes use of source code comment analysis as done in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Wehaibi et al., 2016, Maldonado and Shihab, 2015, Farias et al., 2015, Mensah et al., 2016), it is unique since we contribute with a 6-step prioritization scheme and a text mining algorithm for prioritizing SATD tasks (or *vital few* tasks). Thus, the text mining approach does not only detect SATD tasks as demonstrated in a previous study (Farias et al., 2015) but also prioritize the identified debts based on the prioritization scheme and estimates the rework effort needed with the sole aim of assisting in efficient decision making prior to software release. Again, whilst manual exploration of SATD was performed in previous studies (Potdar and Shihab, 2014, Bavota and Russo, 2016, Maldonado and Shihab, 2015, Wehaibi et al., 2016), we automate the process by introducing a text mining algorithm for SATD analysis. It should be noted that, this study makes use of an effort estimation metric in our recent work (Mensah et al., 2016) to estimate the rework ef-