Reducing Technical Debt: Using Persuasive Technology for Encouraging Software Developers to Document Code

(Position Paper)

Yulia Shmerlin[1], Doron Kliger[2], and Hayim Makabee[3]

[1] Information Systems Department, University of Haifa, Haifa, Israel
[2] Economics Department, University of Haifa, Haifa, Israel
[3] Yahoo! Research Labs, Haifa, Israel
yshmerlin@is.haifa.ac.il, kliger@econ.haifa.ac.il,
makabee@yahoo-inc.com

Abstract. Technical debt is a metaphor for the gap between the current state of a software system and its hypothesized 'ideal' state. One of the significant and under-investigated elements of technical debt is documentation debt, which may occur when code is created without supporting internal documentation, such as code comments. Studies have shown that outdated or lacking documentation is a considerable contributor to increased costs of software systems maintenance. The importance of comments is often overlooked by software developers, resulting in a notably slower growth rate of comments compared to the growth rate of code in software projects. This research aims to explore and better understand developers' reluctance to document code, and accordingly to propose efficient ways of using persuasive technology to encourage programmers to document their code. The results may assist software practitioners and project managers to control and reduce documentation debt.

Keywords: technical debt, documentation debt, documentation, software maintenance, persuasive technology, FBM Model.

1 Introduction

Traditionally, the evolution of software development methods and tools has focused on improving the quality of software systems. The most obvious quality attribute is correctness: the ability of a software system to satisfy its requirements. Correctness is a functional quality attribute, since it relates to the functions performed by the system.

However, software systems should also have several desirable non-functional quality attributes, such as maintainability, extensibility and reusability[3]. These attributes relate to the way a system has been implemented, i.e., to the complexity of the relationships among the modules that compose the system, independently of its correctness. Hence a system is *maintainable* if it may be easily changed, *extensible* if it is easy to add new features and *reusable* if its modules may be easily adopted in new applications.

L. Iliadis, M. Papazoglou, and K. Pohl (Eds.): CAiSE 2014 Workshops, LNBIP 178, pp. 207–212, 2014.

Recently, the metaphor of technical debt has been widely used to describe the gap, both in functionality and quality, between "the current state of a software system and a hypothesized 'ideal' state, in which the system is optimally successful in a particular environment" [4]. One form of technical debt is internal documentation debt [20], i.e., inappropriate, insufficient or non-existing internal documentation. Low-quality documentation is known to affect negatively quality attributes such as maintainability [18][16].

Previous works have identified some of the reasons for poor documentation in software systems. For example, many developers are under-motivated to document, since they perceive writing internal documentation as a secondary task, as opposed to writing the code itself [4]. Moreover, some software development approaches promote the idea that good code should be self-explanatory [17], and therefore comments are not always necessary.

Our goal is to address these and other causes for low-quality documentation, and propose practical solutions that may be adopted to improve this situation. In particular, we believe that a combination of a persuasive technology approach with advanced tool support may transform the nature of internal documentation tasks, in such ways that software developers will choose to adopt them. This paper describes practical experiments that we plan to conduct in order to examine if and how developers can be encouraged to improve documentation and thus reduce documentation debt.

2 Problem Background and Description

Technical debt can be seen as a compromise between a project's different dimensions, for example, a strict deadline and the number of bugs in the released software product. "Shipping first time code is like going into debt. A little debt speeds development so long as it is paid back promptly with a rewrite" [5]. Technical debt is defined as the gap between the current and the ideal states of a software system [1]. This suggests that known defects, unimplemented features and outdated documentation are all considered aspects of debt. Despite the increasing interest in technical debt among both academics and practitioners, this metaphor still lacks a more rigorous and specific definition. Tom et al. [20] identified different elements of technical debt, such as code debt, architecture debt, infrastructure debt, and documentation debt. Devising ways to reduce the latter is the focus of the current study.

Documentation quality has a direct effect on software maintenance. Software maintenance usually refers to the activities carried out after the development completion, and is the most expensive part in the lifecycle of modern software systems [1]. Maintenance includes a broad spectrum of activities, such as error correction, enhancements of capabilities, deletion of obsolete capabilities and optimization. In order to perform these activities effectively, correct and up-to-date technical documentation is required. Outdated or lacking documentation increases the maintenance costs [18].

Yet, currently, lack of proper documentation during development and release of software systems is prevalent [6]. According to Pfleeger [16] 40%- 60% of the maintenance time is spent on studying the software prior to modification because of the lack of appropriate documentation. Additional studies have shown that source

code comments are the most important artifact to understand a system to be maintained [7] and that inline comments can greatly assist in the maintenance work of software engineers [14]. Programs with appropriate documentation were found to be considerably more understandable than those without documentation comments [21].

Despite the importance of documentation, there is evidence in the literature that source code and comments do not evolve in the same rate. Fluri et al. [9] found that newly added code is rarely commented; not all code is commented equally (e.g., the frequency of comments for method calls is much lower than for method declarations); and 97% of comment changes are done in the same revision as the associated source code change. The code evolves in a significantly higher rate than its comments and, as software evolves, it is common for comments and source code to be out-of-sync [13]. In summary, while software engineers may understand the importance of documentation [7], the code is not documented enough in practice [9], [13].

There may be several explanations for this phenomenon. One of the reasons is that documenting is not considered a creative activity, and many engineers prefer solving algorithmic problems instead of writing documentation [4]. Another reason is that many programmers assume that good code is self-explanatory, justifying the lack of documentation [17]. Additionally, since practitioners often work under very strict deadlines, it is easy to leave the documentation behind. Besides, sometimes not documenting can increase job security[8], because it helps programmers to keep an advantage over others and, thus, ensures demand for their services .Finally, the reason may lay in human perception, since software students do not fully understand the need for proper documentation [2].

Recently, several works investigated the investment of companies using agile methods in documentation. The agile manifesto states that direct communication is more valuable than internal documentation [12]. A study of the role of documentation in agile development teams showed that while over 50% of developers find documentation important, or even very important, too little documentation is available in their projects [19].

In conclusion, regardless of the development method used, documentation plays an important role in software products development. Proper documentation drives a more efficient and effective software maintenance and evolution, requiring lower cost and effort. Therefore, it is important to find ways to improve the quantity and quality of comments. To this end, we must find efficient techniques to encourage developers to document their code, thus improving the readability and reducing maintenance time and cost. The objective of this study is to investigate the current state of documentation, and specifically the reasons for developers' reluctance to comment code, and propose a technique to encourage them to document their code, thus decreasing the costs induced by technical debt.

3 Solution Approach

In order to overcome developers' reluctance to document code, we plan to apply the persuasive technology approach. Persuasive technology is an interactive computer technology, which is designed with the goal of reinforcing, changing or shaping people's attitudes or behavior [10]. When a persuasive system is used for

reinforcement purpose, the desired outcome of its use would be to make the users' current behavior more resistant to change. When using the system for changing purposes, the expectation is that the users will alter their attitude or behavior due to the interaction with the system. Finally, when using the system for shaping purposes, successful outcome would be creating a behavior pattern for a specific situation, which did not exist prior to using the system [15].

When designing a persuasive system, it is important to take into consideration the desired influence of the system on its users, since different techniques should be used depending on the desired outcome [15]. In our context, as discussed in the previous section, while programmers are often aware of the importance of documentation, this is not reflected in their behavior.

In order to produce a successful persuasive design, it is important to understand which factors influence behavior. Fogg [10] introduced the Fogg Behavior Model (FBM) for analysis and design of persuasive technologies, which provides a systematic way of studying the factors behind behavior changes. The model implies that behavior depends on the following three factors: motivation, ability, and triggers, each of which has several subcomponents. The motivation factor consists of three core motivators, each of which having two sides: pleasure vs. pain, hope vs. fear and social acceptance vs. rejection. The ability factor represents the simplicity of performing the targeted behavior, and its six subcomponents are time, money, physical effort, brain cycles, social deviance, and non-routine. Finally, the triggers refer to prompts, cues, calls to action, etc. The purpose of a trigger is to signal to the user that the moment for performing the behavior has come. An additional concept of the FBM model is the behavior activation threshold. For a trigger to evoke the desired behavior, a person has to be above that threshold, in a high enough level of motivation and ability.

It should be noted that most of the people are in moderate levels of ability and motivation and effective persuasive system should raise motivation, ability, or both, as well as provide a trigger for the desired behavior. An additional implication of the FBM model is that there exists a trade-off between motivation and ability of performing the behavior, so if we influence at least one factor of this equation, the desired behavior might be triggered [10]. This model has direct implications to our research, since our aim is to propose a system, which will increase performance in the code documentation task, as well as provide a proper trigger in an appropriate time for this behavior to take place.

4 Research Plan and Method

The objectives of our study are to identify the reasons and challenges that impede developers' motivation to document code, and to propose a utility for encouraging documentation and facilitating proper documentation. For this purpose, we will perform two studies: (1) a think-aloud protocol for examining program maintenance tasks performance, and (2) an experiment to assess triggers for documentation.

In the first study we plan to conduct individual think-aloud sessions with about 15 students in their last year of IS undergraduate studies. Each subject will perform a maintenance task, namely, add functionality, to code written and documented by

another student. The purpose is to gain a deeper understanding of the cognitive process a software programmer faces when maintaining code, and specifically while trying to understand the existing code. We will observe to what extent the subject relies on the code documentation during this process, and what are the important features in code comments that help understand existing code.

The objective of the second study is to check whether the use of an existing documentation-triggering tool (CheckStyle:[1] a plug-in for Eclipse IDE) improves documentation. The subjects of this experiment will be first year IS student in a Java course. The students will be divided into three groups: treatment group A will receive the plug-in to activate a module, which will remind them to add comments to the code as they develop it. Treatment group B will receive the same treatment as well as a social motivation – publishing their documentation level status among their peers. The control group will receive the same plug-in with a different module enabled (not related to code commenting). Our hypotheses are as follows:

- H0a: Group A's documentation level will be similar to that of the control group.
- H1a: Group A's documentation level will be higher than that of the control group.
- H0b: Group B's documentation level will be similar to that of Group A.
- H1b: Group B's documentation level will be higher than that of Group A.

The results will be calculated using documentation metrics. In addition, following the experiment, we plan to collect qualitative data via questionnaires with open-ended questions, in order to gain a deeper understanding about the triggers and motivators from the students' perspectives.

Based on the results obtained in these two studies, and additional external validation with professionals from industry, we intend to create a utility, using persuasive technology principles, for encouraging and motivating developers to document their code with proper and contributing comments. The proposed utility will be evaluated and validated with professional software developers.

References

[1] Brown, N., Cai, Y., Guo, Y., Kazman, R., Kim, M., Kruchten, P., Zazworka, N.: Managing technical debt in software-reliant systems. In: Proceedings of the FSE/SDP Workshop on Future of Software Reengineering Research, pp. 47–52. ACM (2010)

[2] Burge, J.: Exploiting multiplicity to teach reliability and maintainability in a capstone project. In: 20th IEEE Conference on Software Engineering Education and Training, CSEET 2007, pp. 29–36 (2007)

[3] Chung, L., do Prado Leite, J.C.S.: On non-functional requirements in software engineering. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Mylopoulos Festschrift. LNCS, vol. 5600, pp. 363–379. Springer, Heidelberg (2009)

[4] Clear, T.: Documentation and agile methods: striking a balance. ACM SIGCSE Bulletin 35(2), 12–13 (2003)

[5] Cunningham, W.: The WyCash portfolio management system. ACM SIGPLAN OOPS Messenger 4(2), 29–30 (1992)

[1] `http://eclipse-cs.sourceforge.net/`

[6] Daich, G.T.: Document Diseases and Software Malpractice. CrossTalk (2002)
[7] De Souza, S.C.B., Anquetil, N., de Oliveira, K.M.: A study of the documentation essential to software maintenance. In: Proceedings of the 23rd Annual Int. Conference on Design of Communication: Documenting and Designing for Pervasive Information, pp. 68–75. ACM (2005)
[8] Drevik, S.: How to comment code. Embedded Systems Programming 9, 58–65 (1996)
[9] Fluri, B., Wursch, M., Gall, H.C.: Do code and comments co-evolve? on the relation between source code and comment changes. In: IEEE 14th Working Conference on Reverse Engineering, WCRE 2007, pp. 70–79 (2002)
[10] Fogg, B.J.: Persuasive technology: Using computers to change what we think and do. Morgan Kaufmann Publishers, Elsevier Science (2003)
[11] Fogg, B.J.: A behavior model for persuasive design. In: Proceedings of the 4th ACM Int. Conference on Persuasive Technology (2009)
[12] Highsmith, J., Fowler, M.: The agile manifesto. Software Development Magazine 9(8), 29–30 (2006)
[13] Jiang, Z.M., Hassan, A.E.: Examining the evolution of code comments in PostgreSQL. In: Proceedings of the 2006 ACM Int. Workshop on Mining Software Repositories, pp. 179–180 (2006)
[14] Lethbridge, T.C., Singer, J., Forward, A.: How software engineers use
[15] Oinas-Kukkonen, H., Harjumaa, M.: A systematic framework for designing and evaluating persuasive systems. In: Oinas-Kukkonen, H., Hasle, P., Harjumaa, M., Segerståhl, K., Øhrstrøm, P. (eds.) PERSUASIVE 2008. LNCS, vol. 5033, pp. 164–176. Springer, Heidelberg (2008)
[16] Pfleeger, S.L.: Software Engineering: Theory and Practice, 2nd edn. Prentice-Hall (2001)
[17] Parnas, D.L.: Software aging. In: Proceedings of the 16th Int. Conference on Software Engineering, pp. 279–287. IEEE Computer Society Press (1994)
[18] Shull, F.: Perfectionists in a world of finite resources. IEEE Software 28(2), 4–6 (2011)
[19] Stettina, C.J., Heijstek, W.: Necessary and neglected? an empirical study of internal documentation in agile software development teams. In: Proceedings of the 29th ACM Int. Conference on Design of Communication, pp. 159–166. ACM (2011)
[20] Tom, E., Aurum, A., Vidgen, R.: An exploration of technical debt. Journal of Systems and Software, 1498–1516 (2013)
[21] Woodfield, S.N., Dunsmore, H.E., Shen, V.Y.: The effect of modulari-zation and comments on program comprehension. In: Proceedings of the 5th Int. Conference on Software Engineering, pp. 215–223. IEEE Press (1981)