Crowdsourcing for API documentation: A Preliminary Investigation

Allahbaksh M. Asadullah
Infosys Labs, Infosys Ltd.
Bangalore, India
Email: allahbaksh.asadullah@infosys.com

Shilpi Jain
FORE School of Management
New Delhi, India
Email: shilpijain@fsm.ac.in

***Abstract*— Developers and researchers have been using crowdsourcing in a variety of fields related to software development and software engineering. Crowd based documentation is another yield of crowdsourcing where the coder community or workers document the software. In the present work, we have analyzed how crowdsourcing can be used for an API documentation. The study is based on the fact that good programmers write descriptive variables and method names and continue to do so for future references. A variety of tools such as Amazon Mechanical Turk, ETurk and DocIt were evaluated for the purpose. Among these, DocIt and ETurk were built in-house. The evaluation of the documentation was performed by experienced coders. This is a preliminary experiment which was performed in a controlled environment. Results were encouraging and help us to determine that in future crowd based documentation might help to reduce time to market and improve software quality.**

Keywords: Crowdsourcing, API Documentation, Amazon Mechanical, Turk, DocIt, E-Turk

I. INTRODUCTION

Development and maintenance of large software systems[1] remains a difficult and daunting problem for any project team. Based on the studies, the percentage of effort goes in requirements phase is 15-20%, analysis and design is 15-20%, development effort is 25-30%, system testing is 15-20%, and maintenance effort across the software development life cycle is typically 5-10%. Until early 90s, in a conventional software development, the modification (adding or deleting a module or functionality) of the code had been a great challenge. This paradigm shifted with the emergence of object oriented programming and open source software development that supports modular programming and library reuse. By breaking down the problem into multiple tasks, different developers[2] can work in parallel. Modular programming allows distributed development that shorten the development and documentation time. Moreover, individual modules are easier to design, implement, and test.

Besides source code, several documents accompany a software system, because there is a possibility that the source code perhaps is not sufficient to convey the objective of the project. Hence, moderate to large sized development projects, irrespective of application, generate a large amount of associated documentation such as documentation of code, algorithms, application programming interface (API), UML diagrams, sequence diagrams, class diagrams, design documents etc. The set of these components are popularly known as software artifacts or technical documents.

Usually, the process of documentation is elaborate and requires a significant amount of effort. A substantiate documentation reduces the maintenance work and further improves the productivity and reusability of the code.

Many developers believe that the documentation doesn't require a high intelligent quotient and it is a waste of time and effort. They consider that the code written by them is sufficient and self-explanatory. On the other hand, another set of programmers appreciate its importance but tend to avoid due to paucity of time or limited resources. Segal [1] observed that professional developers do not volunteer to produce code documents. If necessary, they will write a page or two as a formality. Brief and inappropriate documentation is a matter of concern [2]. To overcome this limitation, new programmers can leverage the abundant repository of unofficial API documents generated by API users on community portals. This unofficial documentation is popularly known as crowd documentation because it is generated from crowdsourcing [3]. These documents have sufficient coverage for practical usages. Blog post and 'stack overflow' are two types of crowd documentation that have highest coverage ratios [4] . Latoza et al. [5] describe the advantages of crowd based development in their work. Crowdsourcing introduces agility in the component development of large-scale industrial projects, especially when dealing with changes in API.

Besides having several benefits of API documentation there are not many tools available in the market which can perform

[1] At Infosys Ltd., any software project's code that exceeds 50k line of code (LOC) falls in the category of large software system (project).

[2] Please note that in the current study, words like developer, coder, programmer, and worker are used as synonyms.

this task efficiently. Hence for the purpose, we have developed two prototypes. In the current study we propose to evaluate and compare the performance of Amazon Mechanical Turk (also known as MTurk) with two in-house developed tools (ETurk and DocIt). The other objective is to test whether they can produce consumable APIs.

II. BACKGROUND LITERATURE

API documentation or Programmers documentation, is a deliverable of technical writing in which a technical writer develops instructions about how to effectively use a software API, hardware (SCPIs) or web-API [6]. In addition to established coders, it is even useful for new coders as it helps them to understand and learn best practices and implementation details. Among many, API documentation is a subset of software documentation. It is often embedded within the source code like Javadoc comments in Java. API documentation is written in plain language which requires a thorough understanding of the API, its arguments, its return type and the languages and interface it supports. The text is often supported by images or hyperlink to other elements of the source code. API usability is important because of the spread of APIs in almost every application domain [7]. Among the factors that affect API usability, is the lack of diverse API documents [8].

The extant literature noticed that unlike open source projects, API documents are rarely updated in industrial setting. In open source development, API documentation for deprecated code is taken seriously and this task is mandatory before they release candidate of the library. A set of specialized tools are available for source code documentation, but rarely used [5]. Typically, software engineers decide on their own what kind of documentation is worthwhile to produce and maintain, and adopt selective tools which help them for the purpose [9] [10] [11]. This has been reported that software engineers tend to ignore complex and time-consuming documentation [11]. Literature shows that scientific documentation does not follow recommended standards proposed by SEBoK (www.sebokwiki.org)

Previous studies have shown that a majority of Java developers prefer to use Javadoc [2] [1] instead of APIs. In another research, Latoza et al. [5] explained how a complex task can be decomposed into set of smaller tasks in crowd development. However, they did not discuss or refer any specific case studies for the same. Kittur et al. [12] showcased how an article writing can be achieved by crowdsourcing. Jiau and Yang [13] studied the API documentation of few open source project like GWT, SWT and Swing. They found that the quality of documentation produced in open source forums was of better quality. To the best of our knowledge, we could not find any research study that specifically talks about the development of API documents through crowdsourcing.

A variety of web platforms are available for software crowdsourcing such as Amazon Mechanical Turk (also known as MTurk). MTurk is one of the sites of Amazon Web Services. It is a crowdsourcing Internet marketplace that enables individuals and businesses to collaborate and perform complex tasks that computers are unable to conclude. A user of Mechanical Turk can be either a "Worker" (employee) or a "Requester" (employer). MTurk is one of the sites of Amazon Web Services. Employers post jobs known as HITs (Human Intelligence Tasks), such as choosing the best among several photographs of a storefront, writing product descriptions, or identifying performers on music CDs. Workers can then browse among existing jobs and complete them for a monetary payment set by the employer. On the basis of jobs performed, Turk creates qualification profile for the workers.

MTurk encounter certain limitations. In MTurk, it was difficult for coders to highlight the syntax and due to which code comprehension was a challenge. Second limitation, the tool is limited to open source applications and doesn't support proprietary software. Therefore, for internal enterprise applications, we developed a similar tool named as E-Turk with enhanced features. The tool was tested with a team of Java coders and it was observed that many developers were still finding it difficult to associate the source code with required class file (which might me parent class, implementations etc.). They suggested that it would be helpful to give information about methods and classes that are doing similar nature of work. Hence we added a feature that can develop a connection between similar classes & methods semantically. The new version was renamed as DociT. To confirm the utility of DociT and E-Turk over MTurk we performed an experiment based study in industrial setting.

III. RESEARCH DESIGN

The research was conducted in two main phases:

Phase1: Preliminary investigation using surveys with software programmers.

Phase 2: Experiment study where software developers wrote API documentation on MTurk and customized prototypes.

Limited time, dynamic requirements, confined research group, and small user base are some of the reasons cited for the absence of documentation [10] [2] [1]. Several researchers raised this concern but how to achieve the goal remain unanswered. To estimate the root cause, in phase I of the research, we conducted a preliminary survey with developer's community in the Silicon Valley of India, i.e. Bangalore, India. The objective was to understand how frequently and precisely our coders document the code, which is the preferred tool for the purpose. The survey was designed by experts which had 15 questions to capture the responses. The survey was sent to approximately 127 Java coders with a minimum coding experience of 3 years and maximum of 5 years. Out of 127, only 95 responded, 2 responses were dropped because of incompleteness and finally 93 responses (63 males and 30 females) constituted the final sample size.

A. Phase I Findings

The results indicated that 46.4% of respondents are in habit to include comments before defining any function, method, class or a variable, while 34.7% do it only when they feel it is needed (when writing a complex algorithm or function), and rest i.e. Approximately 19% do it rarely as they consider that their code is self-explanatory and documenting it is a waste of time. In response to another question, 86% developers reported that they follow proper naming conventions and give meaningful names. These findings led the foundation of our research as our research is based on the fact that programmers write descriptive method and variable names. Further, 83% of developers deliberated that appropriately documented source code facilitate in understanding of the code and leads to reduction in code comprehension time, 33% think documentation writing is a dull & boring job, and 16% of developers think that API documentation is not needed. Since projects have stricter deadlines, their focus remains on adding features and deliver. Almost every coder reported that documentation is a time consuming process and in most of the cases hard deadlines doesn't provide any room for it. Another culprit is 'frequent changes in requirements' which completely shifts the focus of a programmer from documentation.

B. Experiment Study

Mechanical Turk: Mechanical Turk (also known as MTurk) is a decent platform but generic in nature. It lack certain special features. In MTurk, the task submitted as a single HIT is difficult to comprehend because of the code dependency. The HIT is an independent work unit. The main advantage of MTurk is that it offers insights into how one should design the platform, what are the key problem that needs to be addressed which a developer may face.

A CSV file from an open source project, Apache Drill3 was selected to upload in MTurk. The worker had to pass the Java programming test before getting assigned to any live project. Once passed, they attempt to HIT. Every HIT included a fully qualified name of the class, a method and to be documented, and class body. The workers were expected to write the API documentation (java doc) of the method in the text area provided. We have included these three fields so that the workers can understand the package hierarchy from the class name, document the method from the method body provided and use the class body as a reference to identify the relationship of the method from other methods present in the class. Each time the HIT is answered by a specified number of users, the HIT list is updated dynamically with the pool of pending hits. HITs are exhibited to users in random fashion.

Enterprise Turk (E-Turk): E-Turk was designed by mimicking the MTurk and has several modules such as user registration, user modules etc. The user interface of E-Turk is easy to use and almost similar to Mechanical Turk with an exception. Unlike MTurk, in E-Turk, the users were allowed to view all HITs, the source code was rendered with highlighted syntax which makes it easier to comprehend, and submission of the task is easy in comparison to Amazon Mechanical Turk. In MTurk we faced difficulties in posting of certain tasks (HITs) where the code snippet had special characters.

DocIt: This tool was superset of E-Turk. The tool can browse, search and navigate the source code online. Also the tool featured semantic search and gives information on related method and classes. This resembled Eclipse in many aspects.

The UI for DocIt was borrowed from another API explorer tool. When a developer clicks on any previously documented method it gives the latest document which the developer could edit to improve. If there is no documentation for the method then the developer may add the same.

The experiment study with the second platform gave us more insights into the problems faced by the crowd while documenting the source code. For examples, developer could frequently observe the related classes / interfaces. They experienced different patterns in the source code to find out how an object could have created (in case of Factory, Abstract Factory or Prototype Design pattern). These features are fairly common and available in other IDEs. Hence, DocIt was designed to address those key issues.

Prior to the initiation of experiment, certain preparations were made. We followed systems approach and developed a process flow diagram (see Figure 1).

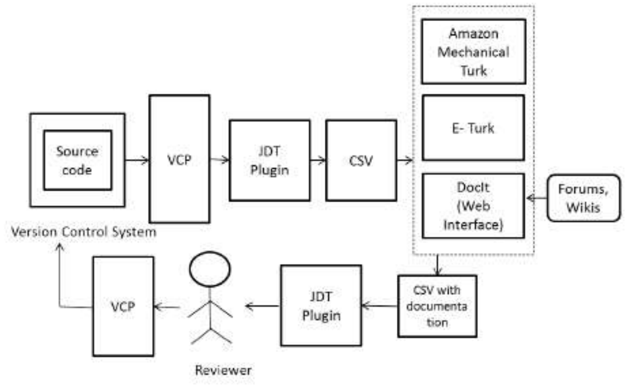

Fig. 1. System Diagram

Version Control Plugin (VCP) – We designed a VCP that extracts latest source code from the version control system. Since Team Foundation Server (TFS) is widely used in the organizations, we designed VCP for the TFS. It is important to note here that one can design or extend VCP for any other version control system by using appropriate libraries such as JHG, JGit etc. This plugin pulls the source code from the

Version Control System which is then be acted by different components of the system.

Source Code Parser – The source code which is pulled from Version Control System is input to the Source Code Parser. The source code parser can parse the elements of the source code. For example, in Java it can tell about the various methods in the class, field variables and their initial values etc. The source code parser analyzes the source code which is a .java file and create an Abstract Syntax Tree (AST). The AST provides detailed view of the source code. The necessary information from AST is then extracted and is saved as Comma Separated File (CSV file) and an index file.

Each .java file created more than 1 row in the CSV file. A row in a CSV file contains details of method. The column attributes are method body, class body, class full name (Package Name +Class Name), which means that for each method (in .java file), we will have a row entry in the CSV file.

The CSV file and the index is input to the systems like Mechanical Turk, E-Turk and DocIt. Each row in the CSV file correspond to one unit of work which is called as HIT (in Amazon Mechanical Turk).

Each HIT is independent of the other HIT and the tasks can be completed without any further detailed knowledge. There is no navigation facility available in Amazon Mechanical Turk and E-Turk. Based on our knowledge and other results, it was realized that navigation plays important role in source code comprehension. Consequently, this limitation was addressed in DocIt. Besides navigability, DocIt has additional functionality. It has rich interface that can browse and explore the source code effectively. DocIt uses two inputs: CSV file and Index. The index feature facilitates navigation.
Below is step by Step process

1. We first extracted the source code elements and related artifacts from version control system (VCS).
2. The source code is then parsed and a CSV File and Index file is prepared.
3. These artifacts are provided to the crowd developers via Mechanical Turk, ETurk, and DocIt.
4. Based on the available source code and other information, developers wrote API documents.
5. The API documents were evaluated by a handful of experienced system architects.
6. Once the API document for particular method is approved, the source code was updated and committed to the version control.

IV. FINDINGS

In case of E-Turk and DocIt we did not conduct any test to check the knowledge of the developer, instead we handpicked a team of coders (Java) from Infosys. These programmers had scored more than 65% of marks in an internal Java exam. An Enterprise Java application was given for the documentation. They were free to choose their tool from the available 3 options. We advertised about these tools through several internal media channels. Nevertheless, MTurk wasn't the preferred choice for documentation. Merely 30% of the HITs were resolved from the given list, and only 23 out of 237 (approx.10%) coders participated in the exercise. These findings were quite surprising hence, we asked the rest for the reasons through a semi structured questionnaire. Their responses are summarized in Table 1.

TABLE I. PHASE I RESPONSES

		Percentage of Responses
1	Lack of experience with MTurk	65%
2	Tool Usability (issues with UI and framework).	72%
3	Complicated HITs	57%
4	Non-challenging HITs / Ambiguous HITs	77%

In case of Mechanical Turk, the workers had no other alternative to visualize additional methods which they can use to document the same class. This means that the person documenting the method will be given a random method mij from a class Ci where mij in Ci. Hence the worker has to read the new class and understand it which was time consuming. The HITs were independent of each other, therefore, there are chances that for every HIT worker may probably get a new class, which leads to waste of time of the worker writing the documentation. HITs are released in certain batches, hence there is no guarantee that all the methods of a single class will be released one at a time. About 60% of coders reported that the HITs did not interest them or motivate to respond.

In E-Turk, the results were comparatively better that MTurk. This could be because of two reasons: a. the coder community, tool, and application were internal resources (i.e. available within the organization setup), b. Since these applications were developed for the client, they were well written and followed industry standards of coding and formatting. All developers were experienced Java coders.

In case of DocIt, a cross reference to the classes and methods was available. This helped the developers in accessing more information about the methods and the classes. They could navigate easily to other classes and methods resulting in improved documentation. DocIt was used by 15 developers. We observed that the developers did navigate the source code and read related classes and methods before documenting the method. Many of the developers looked at the other artifacts which were presented in the tool based on the class the developer was browsing in the tool. For example, for a class called HousingLoan, the developers paid attention towards the related documents and look over the threaded discussions on AutomobileLoan. Almost 45% of the developers resorted to reading these descriptions and found them useful in API document preparations. From the logs of the tool, we found that developers spent a considerable amount of time towards understanding the classes like ILoan (interface), IInterest etc. which were related to the Loan class before writing the API document.

A. Validation of API Documents

Post experiment, a team of system architects and authors of the class files reviewed API documents. The documents were checked for completeness, context and precision. On an average coders wrote 6 lines per method in the API document and reviewers' added 2.5 lines to the submitted API documentation. Three out of four reviewers did spend time to check whether the documentation conveyed intended meaning and relevant to the context. The average time per method spent for review was 7 to 8 minutes. These metrics are decent as successful open source projects follow the same metrics before their source code is committed to any VCS (Version Control System).

Our study was based on the fact that the available tools are insufficient to perform accurate API documentation. For validation, two other source code documentation tools (e.g., TwinText and Doc-O-Matic) were used which comprehend the source code and generates the API documentation. These tools doesn't require any human intervention.

TwinText[3] uses code comments with source code analysis to generate the API documentation. One can define the style of API documentation in TwinText. This tool has a limitation, API documentation produced here embed originator's comments. Some of these comments were generic and written merely for understanding purposes (e.g. the code comments written by the developer merely to understand the internal code structure) which might not be worthwhile for code reader or developer who might use these API in future. The tool does not provide any consumable APIs.

Doc-O-Matic [4] uses the source code as primary artifact and add additional information based on the domain through external inputs. This tools uses Java language semantics to identify the package name, member variables, and method names and then generates the API documentation. We used Doc-O-Matic on the Apache Drill project, and the output (API documentation) was inappropriate. The descriptions had random words and control characters. Our guess is that probably their NLP (Natural language Processor) was unable to generate meaningful sentences.

The objective of current study was to evaluate and compare the performance of DocIt with other similar tools available in the market and whether they can produce consumable APIs. We found that none of the available options could solve the purpose. Manually generated API documents were better and meaningful over automatic API documentation tools. Below are some of the metrics which we picked to check how much time and effort it takes to create API documentation.

V. CONCLUSION AND FUTURE WORK

Our preliminary investigation showed that software API documentation can be achieved by crowdsourcing. A variety of developers review API documentation and prepare the final output through code review system. Prior research confirms that the crowd documentation is dependent upon the paradigm of the programming language. The person writing the documentation should have the knowledge of programming paradigm and the domain of the project. Comparatively, developers spent lesser amount of time and effort in documenting a code where modules are interacting using APIs. We conclude by making an observation that the development of API documentation by crowd sourcing saves time and effort. It further helps the software industry and academia to evolve and generate new software systems and innovate rapidly. We are further exploring additional artifacts that could be helpful in documentation. During the time of writing this paper, we are experimenting with unit test cases and version commit messages. These studies are still in progress.

REFERENCES

[1] J. Segal, "Models of scientific software development," in *1st International Workshop on Software Engineering for Computational Science and Engineering*, 2008.

[2] R. Sanders and D. Kelly, "Dealing with Risk in Software Development," *Software-IEEE,* vol. 25, no. 4, pp. 21-28, 2008.

[3] J. Howe, "The Rise of Crowdsourcing. Wired," *The Wired,* vol. 14, no. 6, 2006.

[4] C. Parnin and C. Treude, "Measuring API Documentation on the Web. , pages 25–30, May 2011.," in *In Proceeding of the 2nd International Workshop on Web 2.0 for Software Engineering*, 2011.

[5] T. D. LaToza, B. W. Towne, A. V. D. Hoek and J. D. Herbsleb, "Crowd Development," in *6th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2013.

[6] G. Singh, "What is API Documentation?," 27 March 2012. [Online]. Available: http://technicalwritingtoolbox.com/2012/03/27/what_is_API_documentation, [Accessed 22 February 2015].

[7] J. M. Daughtry, U. Farooq, J. Stylos and B. A. Myers, "API Usability," in *In Proceedings of the 27th International Conference on Human Factors in Computing Systems: CHI'2009 Special Interest Group Meeting*, 2009.

[8] M. P. Robillard, "What Makes APIs Hard to Learn? Answers from Developers," *IEEE Software,* vol. 26, no. 6, pp. 27-34, 2009.

[9] T. C. Lethbridge, J. Singer and A. Forward, "How Software Engineers use Documentation: The State of Practice," *Software, IEEE,* vol. 20, no. 6, pp. 35-39, 2003.

[10] L. Nguyen-Hoan, S. Flint and R. Sankaranarayana, "A Survey of Scientific Software Development," in *International Symposium on Empirical Software Engineering and Measurement*, 2010.

[3]http://www.ptlogica.com/TwinText/

[4] http://www.doc-o-matic.com/

[11] A. Pawlik, J. Segal and M. Petre, "Documentation Practices in Scientific Software Development," in *5th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, 2012.

[12] A. Kittur, B. Smus, S. Khamkar and R. E. C. Kraut, "Crowdforge: Crowdsourcing Complex Work.," in *24th Annual ACM Symposium on User Interface Software and Technology*, 2011.

[13] H. C. Jiau and F. P. Yang, "Facing up to the inequality of crowdsourced API documentation," in *Sigsoft Software Enfgineering Notes*, 2012.

[14] B. Dagenias and M. P. Robillard, "Creating and evolving developer documentation: Understanding the decisions of open source contributors," in *18th ACM Sigsoft International Symposium on Foundations of Software Engineering*, 2010.

[15] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies: A Survey," in *ACM symposium on Document Engineering*, 2002.

[16] D. Kramer, "API Documentation from Source Code Comments" A Case Study of Javadoc.," in *17th Annual International Conference on Computer Documentation*, 1999.

[17] K. T. Stolee and S. Elbaum, "Exploring the Use of Crowdsourcing to Support Empirical Studies in Software Engineering," in *International Symposium on Empirical Software Engineering and Measurement*, 2010.