Jordan Richards

ENC 1102

7 October 2019

# Research Proposal on Effective Documentation for Open Source Projects

Open source software is arguably the backbone of computer science. By contributing to open source projects, developers are not only able to share their knowledge and skills in a particular area, but also encourage code reuse across organizations, allowing groups to develop products and tools efficiently and with limited resources. In large open source projects, direct communication, whether in person or through virtual meetings or messaging groups, is a lot more limited. Developers and users might not live in the same geographic areas, or might be divided by cultural and language barriers. However, despite these constraints on communication, large open source projects like ReactJs and Spark continue to thrive with thousands of users and contributors. This is due in part to good documentation. In the absence of good direct communication, better documentation is often needed to ensure a project is stable.

Documentation in software engineering serves many purposes, ranging from tracking progress to communicating ideas. Documentation often serves as a source of truth for a project, helping everyone to stay on the same page, to understand the goals and features of each part of the project. Documentation can come in many forms, from in-code comments, to machine generated API listings and pull requests.

While it is easy to define good documentation pragmatically as documentation that accomplishes all the aforementioned goals, the actual characteristics of good documentation are much more loosely defined. By finding patterns in the documentation of popular open source projects, one might hope to better define what makes documentation better able to achieve its goals. One could then use this information to build better documentation writing tools to encourage these practices and help to make their open source project successful. While this study's scope will be narrowed to only analyze open source documentation, some of the findings it uncovers might apply to a much broader range of projects.

## Literature Review

A tutorial on documenting a small open source project is a great place to start to get a handle on some of the most basic forms of documentation for open source projects, and some of the key ideas behind the importance of documenting any project. "A beginners guide to documentation" is a great article that provides a loose guide for how to write a README, an introductory document intended to get new users familiar with your project. One key idea presented within this article is that without information about what your project is and how to use it, those unfamiliar are very likely to find an alternative. This suggests that there is a direct correlation between the success (the widespread usage) of a project, and the quality of its documentation.

The ways in which documentation is created and used within a project is often directly related to how the project is structured. Two popular project structures, Agile and Waterfall offer seemingly conflicting views on the role of documentation in a project. While not specifically applicable to an open source format, the ideas presented by these two project structures are still applicable. In "Managing the Development of Large Software Systems", Royce suggests that in projects with long timelines, or with large scales, verbal communication is often not enough to make progress. By writing documentation, the various members of a project don't have to rely on direct interaction with previous members in order to be effective (Royce). In contrast, Agile places a higher importance on maintaining direct communication, and suggests that documenting the development practice should not considered as "first-class" (Cohn and Ford 2003). In the study "Introducing an agile process to an organization", Cohn and Ford found that smaller team sizes and direct communication between team members -- facilitated by proximity -- decreased the chances of failure. While the role of documentation is not the same in each of these models, it does suggest that effective documentation is important when communication is limited, as is the case in most open source projects.

While several articles suggest that documentation should be a priority in open source projects, not many provide specific details on what qualities an open source project should have. However, in particular cases strong guidelines for certain types of documentation do exist. For instance, the pull request guide presented by the Linux Foundation. As one of the first projects with a modern open source model that is still active today, the documentation of the Linux kernel provides a base for which other projects might build upon. "Submitting patches: the essential guide to getting your code into the kernel" provides strict guidelines for submitting pull requests for Linux. Most of the guidelines presented reflect the hierarchical trust model of the Linux kernel, where a developer would include information about the code they are commiting in patch notes to encourage a trusted maintainer to incorporate their changes into a formal pull request. These maintainers would then work together to incorporate these larger pull requests into the central repository of Linux. While this model works for Linux, other open source projects have adapted this model to their own needs, and their documentation should reflect that.

## Theoretical Framework

Documentation in software development projects forms a genre set, a set of formats and writing styles that help an author achieve a particular goal. Often, genres are associated with discourse communities, and the transfer and accumulation of knowledge within a discourse community helps to shape its genre set. For instance, the formatting of a document might be chosen to express information that the discourse community as a whole finds useful. Additionally, a document might choose to omit certain information, especially background information, to more efficiently convey information to fellow members of the community. As these genres develop, one can use trait analysis in order to gain insight about how each trait makes documentation useful to software developers.

The primary framework that will be used to guide this study is Grounded Theory. Developed by Strauss and Graser, Grounded Theory is a strategy for taking in data, encoding it, and efficiently finding patterns for further analysis. Grounded Theory progresses in a few phases, beginning with the coding phase, where key terms and ideas are extracted. These terms are then used to describe and condense each sample into its components in the memoization phase. These memos are then sorted, at which point pattern finding can begin. This framework provides a well defined structure for use while analyzing large text samples, reducing the bias inherent when doing qualitative data analysis and allowing meaningful conclusions to be drawn.

The extracted key terms should help to characterize the discourse community surrounding a project, separating members into distinct groups based on the nature of their interactions with the documentation, code, and other observed artifacts. Each group would then play a role in the development of the project. For instance, a particular group of users might not contribute directly to the codebase, but might leave feedback on a particular issue on Github, referring back to other documentation as a source of truth. During the coding phase, some focus should be put on how documentation facilitates these types of communication. These data points will help us during the sorting phase, allowing us to draw conclusions about the genre set as a whole.

One of the initial assumptions going into this is that in an open source setting, direct communication with prospective users is not usually possible, so popular open source repositories must have effective documentation to convince developers to take an interest. This study takes advantage of this to find samples representative of effective documentation.

## Research Questions

This study has one main research question: What common traits of good documentation should developers aim to reproduce in their own open source projects? As mentioned before, the scope of the question is limited to open source projects, but the findings might be applicable to other types of projects, even extending outside of software engineering.

## Methods

First, a sample of popular repositories is chosen to get our data. A good place to do that is on Github, a platform currently home to the largest share of public repositories. Relying on the assumption that popular projects are successful, and therefore must have good documentation, we can select these repositories based on the Github "star" system. Users of the platform can star a repository if they are interested, providing an indicator to other visitors that a project might be worth using. A sampling of projects could be chosen on the basis of the total number of stars, as well as from the list of trending Github repositories, to avoid bias towards longer-running projects.

After selecting these projects, actual data needs to be scraped. This can be done by hand or via scripts, but documentation should be extracted from a few key areas for each repository: code comments, issues, pull requests, and the project wiki / external documentation (if available). In line comments should be extracted with a few lines of code for context. External documentation might not be present for every repository, but it is a primary source of documentation. While collecting this data, it is important to organize it by repository and category of origin, for use in the sorting process later.

Now that there is data to process, the coding phase begins. Following the Strauss and Glaser method, key words and ideas should first be extracted from the corpus of text. Afterwords, these key ideas can then be used to create memos for each sample, again taking note of the origin of each one. After the memoization phase will be a sorting phase. In this case, it would be most appropriate to sort the data into a table of two axes: one axis representing the origin category (comment, issue, pull request, etc.) and another representing the presence of a key idea from the coding phase.

Now that the data has been memoized and sorted, the data can be systematically searched for patterns in the data. In essence, one would like to keep track of the set of patterns they have already identified in the data, continuously adding new patterns not already present in the set. From this set of patterns, one can draw conclusions about the genres of documentation.

References

Holscher, E. (2016, May 12). A beginner's guide to writing documentation. Retrieved from https://www.writethedocs.org/guide/writing/beginners-guide-to-docs/

Linux Foundation. (n.d.). Submitting patches: the essential guide to getting your code into the kernel. Retrieved from https://www.kernel.org/doc/html/v4.12/ process/submitting-patches.html

Royce, W. W. (n.d.). Managing the development of large software systems: concepts and techniques. ICSE '87 Proceedings of the 9th International Conference on Software Engineering, 328–338. Retrieved from http://www-scf.usc.edu/~csci201/lectures/Lecture11/royce1970.pdf

Cohn, M., & Ford, D. (2003). Introducing an agile process to an organization. Computer, 36(6), 74–78. doi: 10.1109/MC.2003.1204378

Forward, A., & Lethbridge, T. C. (2002). The Relevance of Software Documentation, Tools and Technologies: A Survey. Proceedings of the 2002 ACM Symposium on Document Engineering, 26–33. Retrieved from http://www.literateprogramming.com/doceng.pdf

Glaser, B., & Strauss, A. (1967). Discovery Of Grounded Theory: strategies for qualitative research. Routledge.