

Jordan Richards

ENC 1102

5 November 2019

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 be 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 committing 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. Afterwards, 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.

In order to simplify the web-scraping and coding process, a script was written to generate a summary document including a random sample of a few features from a given repository.

While such a summary might not be complete, the key elements we pick up on will help to guide how additional data will be collected to supplement the data collected in the sample.

The choice of features to extract in the script, therefore, should provide an effective overview of a repository, but some sacrifices could be made to limit the scope of pages to scrape. For instance, the script does not identify and scrape external documentation sites as there was no consistent methods to do so. In the end, the set of features chosen for the initial sampling of each repository included markdown documents, inline comments, issues, and pull requests. These categories are present in all of the top repositories, play a key role in the contribution and usage of an open source project, and allow us to automate the scraping process.

Access to the data collected in this study is available via a Github repository [See the Appendix]. The repository includes the script that was used to scrape the repository, the reasoning behind the selection of the top 10 repositories, and the summary documents for each of the sampled repositories. This provides the tools for reproducing this study with a different sample of repositories. The documents with codes in the margins will also be made available once the analysis is complete.

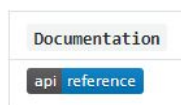
Results

The first set of documents fall under the markdown category. These rich text documents are the most flexible of any scraped category of documents, serving a variety of purposes from *README* files to API documentation. This flexibility can be attributed to the wide range of

features markdown provides, allowing developers to present complicated ideas effectively at minimal cost. An example of a markdown document, is shown below to give an example of some of the types of content that can be incorporated in the format.



TensorFlow



[TensorFlow](#) is an end-to-end open source platform for machine learning. It has a comprehensive, flexible ecosystem of [tools](#), [libraries](#), and [community](#) resources that lets researchers push the state-of-the-art in ML and developers easily build and deploy ML-powered applications.

TensorFlow was originally developed by researchers and engineers working on the Google Brain team within Google's Machine Intelligence Research organization to conduct machine learning and deep neural networks research. The system is general enough to be applicable in a wide variety of other domains, as well.

TensorFlow provides stable [Python](#) and [C++](#) APIs, as well as non-guaranteed backward compatible API for [other languages](#).

Keep up-to-date with release announcements and security updates by subscribing to announce@tensorflow.org. See all the [mailing lists](#).

Install

See the [TensorFlow install guide](#) for the [pip package](#), to [enable GPU support](#), use a [Docker container](#), and [build from source](#).

To install the current release for CPU-only:

```
$ pip install tensorflow
```

Use the GPU package for [CUDA-enabled GPU cards](#) (*Ubuntu and Windows*):

```
$ pip install tensorflow-gpu
```


Figure 1. A snippet of the heading of the Tensorflow README page.

Markdown documents in a repository can be split into a few subcategories based on their usage. These documents are often ordered in some form of hierarchy, rooted at the README file for the repository as a whole, then branching into a series of other files providing more detailed descriptions of different parts of the project. A notable example of this is in the VueJs repository, where the project is broken into packages and each package has its own README. There are several elements common across the subcategories. For instance, most categories begin with a description of the topic the markdown file is meant to present, often as one of the first lines. markdown files also often provide examples, consisting of code snippets or diagrams, helping to better illustrate key ideas. Markdown files also take advantage of links to external pages, such as a Wiki, forum, or external documentation, often presenting information in more elaborate formats not supported by markdown, or to avoid tangents. Other common elements include API references and developer tutorials. In general, markdown documents in several repositories followed a simple pattern: start with a description, followed up by a list of features, each illustrated by an external link, diagram, code snippet, or other visualization.

A key subcategory of markdown documentation is the contribution guideline. These guidelines provide information to a developer about how to format other forms of documentation, such as pull request, commit messages, and issues. This information is often supported by templates, providing a specific format to ensure consistency of code and documentation across the repository. An example contribution guideline is shown below.

Vue.js Contributing Guide

Hi! I'm really excited that you are interested in contributing to Vue.js. Before submitting your contribution, please make sure to take a moment and read through the following guidelines:

- [Code of Conduct](#)
- [Issue Reporting Guidelines](#)
- [Pull Request Guidelines](#)
- [Development Setup](#)
- [Project Structure](#)

Issue Reporting Guidelines

- Always use <https://new-issue.vuejs.org/> to create new issues.

Pull Request Guidelines

- The `master` branch is just a snapshot of the latest stable release. All development should be done in dedicated branches. **Do not submit PRs against the `master` branch.**
- Checkout a topic branch from the relevant branch, e.g. `dev`, and merge back against that branch.
- Work in the `src` folder and **DO NOT** checkin `dist` in the commits.
- It's OK to have multiple small commits as you work on the PR - GitHub will automatically squash it before merging.
- Make sure `npm test` passes. (see [development setup](#))
- If adding a new feature:
 - Add accompanying test case.
 - Provide a convincing reason to add this feature. Ideally, you should open a suggestion issue first and have it approved before working on it.
- If fixing bug:
 - If you are resolving a special issue, add `(fix #xxxx[,#xxxx])` (`#xxxx` is the issue id) in your PR title for a better release log, e.g. `update entities encoding/decoding (fix #3899)`.
 - Provide a detailed description of the bug in the PR. Live demo preferred.
 - Add appropriate test coverage if applicable.

Development Setup

You will need [Node.js version 8+](#), [Java Runtime Environment](#) (for running Selenium server during e2e tests) and [yarn](#).

Figure 2. A snippet from the top of the Vue.js Contribution Guide.

Inline comments provided another source of documentation. The scope covered by inline comments was far more limited than the markdown documents, often falling into one of two categories with significant overlap between them: comments explaining flow, and comments explaining technical details. Comments that explained flow allow developers already familiar at a high level with the code's function to more easily follow along during a read through. While design paradigms such as functions and abstraction already provide some level of logical organization, flow comments are often needed to supplement in longer functions where the purpose of some lines might be unclear. The comments that explained technical details covered other cases when low level technical details were needed to provide clarification for why code was implemented in a certain way. Both types of comments are often sparse, often spanning only a line or so, with few words, making the code easier to read as more of it fit can fit onto a single screen. An example of an inline comment is shown below:



```
if (level < 0)
    level = MZ_DEFAULT_LEVEL;
if ((level & 0xF) > MZ_UBER_COMPRESSION) {
    // Wrong compression level
    printf("%s", "Wrong compression");
    goto cleanup;
}
```

Figure 3. An example of an inline comment explaining the flow of the code.




Issues and pull requests are the only types of documentation generated without modifying the source of the repository. While the other forms of documentation involve modifying the “ground truth” of the repository, issues and pull requests provide a history of the changes to the

repository, new information is only intended to be added to the record, rather than modifying and updating existing records. An example snippet of a pull request is shown below.



Update Fixtures to use new APIs #17380



 Open sebmarkbage wants to merge 1 commit into facebook:master from sebmarkbage:fixfixture 



Conversation 2 Commits 1 Checks 0 Files changed 14 +52 -18



 sebmarkbage commented 2 hours ago Member  




Also renamed unstable-async to concurrent.

  sebmarkbage requested review from bvaughn, gaearon and acdlite 2 hours ago

  facebook-github-bot added the **CLA Signed** label 2 hours ago

  Update Fixtures to use new APIs ... 4ede46b

  sebmarkbage force-pushed the sebmarkbage:fixfixture branch from e13adb6 to 4ede46b 2 hours ago



 codesandbox bot commented 2 hours ago  




This pull request is automatically built and testable in [CodeSandbox](#).

To see build info of the built libraries, click [here](#) or the icon next to each commit SHA.







Latest deployment of this branch, based on commit [e13adb6](#) :

Sandbox	Source
determined-colden-i1ph1	Configuration

  gaearon approved these changes 2 hours ago View changes

 codesandbox bot commented 2 hours ago  

Reviewers

-  gaearon 
-  bvaughn 
-  acdlite 

Assignees

No one assigned

Labels

CLA Signed


Projects

None yet

Milestone

No milestone

Notifications Customize

 **Subscribe**

You're not receiving notifications from this thread.

3 participants




  

Figure 4. A snippet of the heading of a pull request.

Focusing on issues, most issues follow a set schema. Users that start an issue often begin with a general description of the bug they are experiencing, followed by formatted steps for reproducing the bug. This format is standardized through the use of issue templates, and

regulated through the use of an automated system that flags and closes issues that don't adhere to the template. Developers or maintainers contributing to the repository might then join the conversation, offering solutions or closing the issue if it is off topic. Often bugs are false positives, resulting in a developer clarifying a feature. Sometimes, an issue will be closed without solving the problem as they are not clearly formatted or worded, or are seen as off-topic. This was reflected in the other documentation of the repository, where Issue guidelines specifically mention ways to improve the clarity of a bug report, as well as key things to avoid. As a whole, issues provide documentation for the project, providing other developers more information about unclear features, commonly asked questions, as well as information about any ongoing issues that they might contribute back to.

Discussion

One important overarching idea presented by other writing in the area of software documentation, was the importance of documentation in attracting other developers to use and contribute back to the project. From this perspective, some features should exist across our genre set that help to advance this cause. One instance of such a feature is the hierarchical organization of markdown documents. By including documentation explaining the function of the files in a particular folder at each level, developers help developers unfamiliar with the history of the repository to understand its structure more easily. Another feature reflecting this might be the concise nature of inline comments within the code itself. By restricting explanations to short clarifications of confusing sections of the code, inline comments can improve clarity without

sacrificing readability. All of the genres in the set reflect have some features that support on-boarding new developers.

Many of the other features exhibited by the documents also contribute to the maintenance of the existing community of developers working on the project. For example, many of the elements highlighted in the contribution, bug reports, feature requests, and pull request guidelines seek to address common issues in these processes. A particular example of this presented in several of the bug report guidelines analyzed, was the requirement to adhere to a particular format within the bug report. The stated purpose for in multiple repositories was to ensure that developers had enough information to understand the bug to dismiss false positives (user error) as early as possible without having to follow up with the original poster. This suggests that contributors to a repository take part in a continuous process where new bugs and feature requests are made, and the ideas learned -- whether it be information about a technical problem or information about issues with the process itself -- are incorporated into the documentation. In this way, the updates developers make to guidelines for future documentation are just as important as the technical ideas they wished to develop in the first place.

All of these features work together to augment the limited communication between developers characteristic of an open source project. Referring back to the findings of Royce in the Waterfall paper and Cohn and Ford's article about integrating Agile into a work environment, we see in two unique project structures the tradeoff between having high frequency communication (such as in Agile) in lieu of more detailed, monolithic documentation (as is encouraged in the Waterfall paper) is consistent. Perhaps, in an open source environment where

direct communication is inconsistent or impossible between developers, more flexible forms of documentation (e.g. Markdown) were developed to address those issues.

Conclusions

The two processes described in the previous section -- one attracting new developers, the other maintaining the community of existing developers -- are the impetus behind a repositories success. These mechanisms work together to create repositories with the features exhibited in the data; features supporting clearer documentation and maintainable contributions. By focusing on reproducing these mechanisms, other open source projects can gain success through developer contribution. Further investigations might use this model to better narrow the search and more accurately identify the specific features that lead to a repositories success, or perhaps to gain further insights into how this model might generalize to other software development projects outside of open source.

Appendix

Project Repository: http://github.com/thefinalstarman/research_scraper/

Includes all the scraped data for this project, as well as the script to allow scraping new repositories.

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.
- Tensorflow README [Screenshot]. (Retrieved November 2019). <https://github.com/tensorflow/tensorflow>

VueJs Contribution Guide [Screenshot]. (Retrieved November 2019).

<https://github.com/vuejs/vue/blob/dev/.github/CONTRIBUTING.md>

Vlang zip.c [Code Snippet]. (Retrieved November 2019).

<https://github.com/vlang/v/blob/master/thirdparty/zip/zip.c>

Facebook/React “Update Fixtures to use new APIs” [Screenshot]. (Retrieved November 2019).

<https://github.com/facebook/react/pull/17380>