Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors

Barthélémy Dagenais and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{bart, martin}@cs.mcgill.ca

ABSTRACT

Developer documentation helps developers learn frameworks and libraries. To better understand how documentation in open source projects is created and maintained, we performed a qualitative study in which we interviewed core contributors who wrote developer documentation and developers who read documentation. In addition, we studied the evolution of 19 documents by analyzing more than 1500 document revisions. We identified the decisions that contributors make, the factors influencing these decisions and the consequences for the project. Among many findings, we observed how working on the documentation could improve the code quality and how constant interaction with the projects' community positively impacted the documentation.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Documentation, Experimentation, Human Factors

1. INTRODUCTION

Developers usually rely on libraries or application frameworks[1] when building applications. Frameworks provide standardized and tested solutions to recurring design problems. For example, hundreds of applications like Google Code Search and Twitter use the JQuery framework to provide an interactive user experience with Javascript and AJAX.[2]

To use a framework, developers must learn many things such as the domain and design concepts behind the framework, how the concepts map to the implementation, and how to extend the framework [12]. Various types of documents are available to help developers learn about frameworks, ranging from Application Programming Interface documentation to tutorials and reference manuals.

We find though that very little is known generally about the creation and maintenance of developer documentation. For example, the Spring Framework manual[3] has approximately 200 000 words (twice the size of an average novel) and has gone through five major revisions. Creating and maintaining this documentation potentially represents a large effort yet we do not know the kind of problems documentation contributors encounter, the factors they consider when working on the documentation and the impact their documentation-related decisions have on the project. For instance, does documenting a change immediately after making it have different consequences than documenting all changes before a release? Answering these questions provides insights about the techniques that are needed to optimize the resources required to create and maintain developer documentation.

We conducted an exploratory study to learn more about the documentation process of open source projects. Specifically, we were interested in identifying the documentation decisions made by open source contributors, the context in which these decisions were made, and the consequences these decisions had on the project. We performed semi-structured interviews with 22 developers or technical writers who wrote or read the documentation of open source projects. In parallel, we manually inspected more than 1500 revisions of 19 documents selected from 10 open source projects.

Among many findings, we observed how updating the documentation with every change led to a form of embarrassment-driven development, which in turn led to an improvement in the code quality. We also found that all contributors who originally selected a public wiki to host their documentation eventually moved to a more controlled documentation infrastructure because of the high maintenance costs and the decrease of documentation authoritativeness. Such observations could enable practitioners to make informed decisions by analyzing the trade-offs encountered by their peers and researchers to build documentation tools that are adapted to the documentation process.

2. METHOD

We based our exploratory study on grounded theory as described by Corbin and Strauss [5]. Grounded theory is a qualitative research methodology that employs *theoretical sampling* and *open coding* to formulate a theory "grounded" in the empirical data. By following grounded theory, we

[1]Unless otherwise specified, we use the term framework to represent any reusable software artifacts such as libraries and toolkits.

[2]http://docs.jquery.com/Sites_Using_jQuery

[3]References to project and documentation tools are presented in Table 5 in the Appendix

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
FSE-18, November 7–11, 2010, Santa Fe, New Mexico, USA.
Copyright 2010 ACM 978-1-60558-791-2/10/11 ...$10.00.

started from general research questions and refined the questions, and the data collection instruments, as the study progressed. As opposed to random sampling, grounded theory involved refining our sampling criteria throughout the course of the study to ensure that the selected participants were able to answer the new questions that have been formulated. For example, after having interviewed two contributors of Perl projects, we filtered out further Perl projects; after having interviewed four contributors from library projects, we sent more invitations to contributors of framework projects.

We analyzed the data, collected through interviews and document revisions, using open coding: we assigned codes to sentences, paragraphs, or revisions and we refined them as the study progressed. We then reviewed the codes several times and linked them to emerging categories, a process called axial coding. Finally, the goal of a study using grounded theory is to produce a coherent set of hypotheses laid in the context of a process, that originates from empirical data. Although all reported observations are linked to specific cataloged evidences, we elide some of these links for the sake of brevity.[4]

Our method follows that of previous software engineering studies based on grounded theory [1, 8, 9]. These references provide an additional discussion on the use of grounded theory in software engineering.

2.1 Data Collection

We learned about the documentation process of open source projects by gathering data from three sources. We interviewed developers who contributed to open source projects and their documentation (the *contributors*): these developers were often the founder or the core maintainer of the project.[5] Most of the observations reported in this paper come from these interviews. We also interviewed developers who frequently used open source projects and who read their documentation (the *users*). We wanted to determine how developers used documentation and what kind of documentation was the most useful to them. Finally, we analyzed the evolution of 19 documents from 10 open source projects (the *historical analysis*). Because some projects started more than 15 years ago, it was often difficult for the participants to remember the various details of the documentation process. Our systematic analysis of the revisions provided us with a more comprehensive and detailed view of that documentation's evolution.

The projects of the contributors, the users, and the historical analysis were selected in parallel so they are not necessarily the same. We used this strategy to preserve the anonymity of the contributors and to allow us to provide concrete examples by naming real open source projects when discussing observations from the users' interviews and the historical analysis. Additionally, this sampling strategy enabled us to perform data triangulation by evaluating our observations on different projects.

The Contributors. To recruit contributors, we began by making a list of open source projects that were still being used by a community of users and that were large enough to require documentation to be used. We only selected projects that fulfilled these five criteria:

Part. Code	(years) Exp.	Project Domain	Prog. Lang.
U1	> 10	Web Applications	Java, PHP
U2	> 10	System Prog., Database	Perl
U3	> 20	System Prog.	C
U4	> 10	System Simulators	C,C++,Java
U5	> 5	Web Applications	Python, Java, C
U6	> 5	Financial Applications	Java
U7	> 5	Web Applications	PHP
U8	> 25	Database	C++
U9	> 25	Web Applications	PHP
U10	> 3	Web Applications	PHP

Table 1: Documentation users

1. The project offered some reuse facilities for programmers (e.g., frameworks, libraries, toolkits, extensible applications),
2. The project was more than one year old.
3. There was at least one active contributor in the last year (e.g., a contributor answered a question on the mailing list in 2009).
4. The project had more than 10k lines of source code.
5. The project had more than 1000 users (measured by the number of downloads, issue reporters, or mailing list subscribers).

We selected projects from a wide variety of application domains and programming languages to ensure that our findings were not specific to one domain in particular.

After having selected a project, we looked at its web site and at the source repository to identify the main documentation contributors. When in doubt, we contacted one of the founders or core maintainers. We sent 49 invitations to contributors, 12 of which accepted to do an interview.

Each contributor who accepted our invitation participated in a 45-minute semi-structured phone interview in which we asked open-ended questions such as "how did the documentation evolve in your project?" and "what is your workflow when you work on the documentation?".

A few contributors talked about various projects they worked on or used, but most contributors focused on one project. The programming language of the projects varied greatly: Perl (2 contributors), Java (2), Javascript (1), C(2), C++ (1), PHP (2), Python (2). The age of the projects ranged from 1.5 years to more than 15 years with an average of 8.7 years. The application domains were also varied: programming language library (4), database or databinding library (3), web application framework (3), blogging platform (1), and web server (1). Finally, all of our participants had more than five years of programming or technical writing experience (up to 25 years).

The Users. To recruit developers who used open source projects and read documentation, we relied on the list of users of stackoverflow.com, a popular collaborative web site where programmers can ask and answer questions. We wanted to interview users who had various amounts of expertise in terms of programming languages and years of programming experience. Stackoverflow user profiles indicate how many questions each user has asked and answered and the tags associated with these questions (e.g., a question might be related to *java* and *eclipse*). We filtered out all users who did not have contact information published on their profile and who were primarily answering questions related to the

[4]Specific evidence will be provided upon request.

[5]Unless otherwise specified, we assume that the contributors have commit access to their project's repository.

Project	Prog. Lang.	Domain	Document	Length Words	Age Yrs	#CS	#C	%CC
Django	Python	Web Fmk.	Tutorial Part 1	3700	4.25	89	11	31%
			Tutorial Part 3	2692	4.25	61	7	57%
			Model API	4140	4.25	191	7	53%
WordPress	PHP	Blogging Platform	Writing a Plug-in	2523	4.00	126	56	wiki
			Plug-in API	2013	4.75	127	56	wiki
KDE Plasma	C++	GUI Fmk.	Getting Started	1521	2.00	51	21	wiki
			Plasma DataEngines	1854	0.75	10	7	wiki
Hibernate 3	Java	Databinding Fmk.	QuickStart	2497	1.00	21	2	9%
			Collections Mapping	3076	1.00	37	4	11%
Spring	Java	Application Fmk.	Beans Framework	30061	4.50	233	15	18%
			Transactions	9584	5.75	87	9	26%
GTK+	C	GUI Fmk.	GTK+ 2.0 Tutorial	56765	9.00	54	10	28%
Firefox	XML	Web Browser	How to build an extension	3163	4.25	316	143	wiki
DBI	Perl	Database Lib.	Module Documentation	34221	5.00	145	3	19%
Shoes	Ruby	GUI Fmk.	Manual	18887	1.00	34	5	6%
Eclipse	Java	Application Fmk.	Creating the plug-in project	559	4.75	16	6	25%
			Application Dialogs	720	7.25	26	8	4%
			Documents and Partitions	841	6.00	24	8	8%
			Resources and the file system	1638	7.75	26	8	8%

Table 2: Evolution of documents

.NET platform because we judged that they were less likely to have a rich experience with open source projects.[6]

We sent 38 invitations and recruited 10 participants. We sent each participant an email asking for a list of open source projects that had good or bad documentation. We purposely did not define good or bad documentation because we wanted the participants to elaborate on their definition during the interview. Each developer participated in a 30-minute semi-structured phone interview that focused on their experience with the documentation of the projects they selected, and then, on their experience with documentation in general.

Table 1 shows the profile of the developers we interviewed: the number of years of programming experience, the main field they are professionally working in, and the programming languages they mentioned during the interview. Most participants used many open source projects as part of their work or as part of hobby projects so their documentation needs are not exclusive to their field of work.

The Historical Analysis. We systematically analyzed the evolution of documents of open source projects that maintained their documentation in a source repository (e.g., CVS) or in a wiki. We also used the same criteria as for the contributors to select projects for our historical analysis.

For each project, we selected from one to four documents. The first document was a tutorial or a similar document that told users how to get started with the project. The second document was a reference (e.g., list of properties). We assumed that these two types of documents were distinct enough that they might exhibit different evolution patterns. We had to analyze a different number of documents per project because there is no documentation standard across projects and it was impossible to compare documents of the same size or of the same nature. For example, documents ranged from a complete manual in one file (e.g., the GTK Tutorial) to document sections separated in small files and presented on many pages (e.g., Eclipse help files).

We analyzed the history of the documents by looking at their change comments and by comparing each version of the documents. This was necessary because often the change comment was not clear enough. For example, a commit comment mentioned fixing a "typo", but in fact, the actual change shows a code example being modified. Through several passes of open coding, we assigned a code to each revision to summarize the rationale behind the change. Table 2 shows descriptive statistics of the documents we inspected such as the time between the first and last revision *that we could find* (in years), the number of change sets (#CS), the number of different committers who modified the files (#C), and the percentage of revisions that originated from community contributions (%CC). We report the details of the revision classification in Appendix A.

We considered that all revisions that mentioned a bug number, a contributed patch, or a post from a forum or a mailing list originated from the community. It was not always possible to determine the source of the change when the documents were hosted on a wiki, so we indicated "wiki" in the table.[7]

3. CONCEPTUAL FRAMEWORK

Following the analysis of the interviews and the document revisions, we identified three *production modes* in which documentation of open source projects is created. Although we expected documentation to be produced in different modes, the study helped us concretize what these modes were and what they corresponded to in practice. These production modes guided our analysis of the main decisions made by contributors (Section 4). Figure 1 depicts how the documentation effort was distributed in the lifecycle of the open source projects we studied. First, contributors create the *initial documentation*, which requires an upfront effort that is higher than the regular maintenance effort. Then, as the

[6]The documentation experience of .NET developers is of interest, but not for this particular study on open source projects. We are aware that with the CodePlex project (www.codeplex.com), open source projects in .NET are becoming more mainstream.

[7]This is only a rough estimate because core contributors sometimes create bug reports themselves and other times, they forget to include the source of the change request.

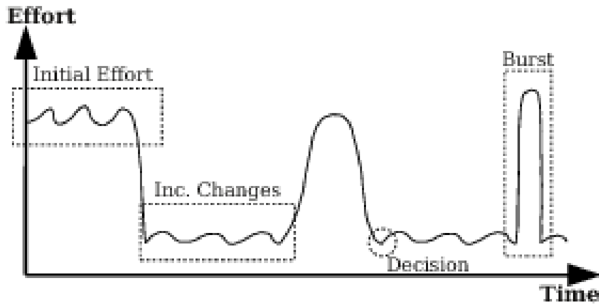

Figure 1: Documentation production modes

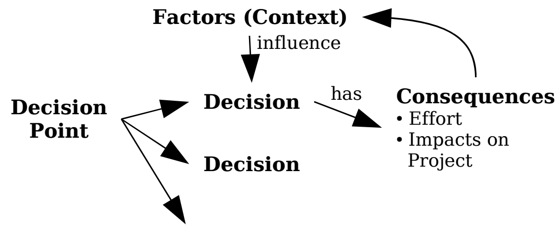

Figure 2: Decisions made in a documentation production mode

software evolves, contributors *incrementally change* the documentation in small chunks of effort (e.g., spending 20 minutes to clarify a paragraph). Sometimes, major documentation tasks such as the writing of a book on the project requires a *burst* of documentation effort.

In addition to the three production modes, we note that documentation writers make important *decisions* at specific *decisions points*. As illustrated in Figure 2, decisions are influenced by contextual factors and they have consequences in terms of required effort and impacts for the project. This paper focuses on the relationships between the decisions, their factors, and their consequences.

For example, for the decision point "When to adapt the documentation to the project's evolution", there are many possible decisions (e.g., updating the documentation shortly after making a change, before an official release, before making a change, etc.). The decisions related to a decision point are not mutually exclusive, but each decision has some specific effort and impact associated with it. The consequences of a decision can also become a factor over time. For example, four contributors sought to document their changes as quickly as possible after realizing that they often improved their code while documenting. We analyzed the consequences of the documentation decisions from many perspectives (contributors, users, and evolution) to evaluate the trade-offs involved with each decision.

4. DECISIONS

We provide an overview of the documentation production modes and the decisions points. Then, we discuss in detail the six decisions that had the largest impact on the documentation creation and maintenance of the projects we studied, as determined by our analysis. Underlined sentences represent major observations for each decision. Table 3 provides a summary of the consequences of these six decisions on five aspects of open source projects.

Initial Effort. When a project starts, contributors encounter two main decision points. First, contributors must select tools to create, maintain, and publish the documentation. There are three main types of infrastructure that are used by contributors, sometimes in combination with each other: wikis (see Section 4.1), documentation suites (e.g., POD, Sphinx, or Javadoc), and general documents such as HTML. In our historical analysis, we observed that the editing errors (e.g., forgetting a closing tag) caused by the syntax of any documentation infrastructure were responsible for an important amount of changes and that better tool support could probably mitigate this problem: 55.4% in Eclipse (HTML), 11.4% in Django (Sphinx), 11.1% in GTK (SGML), and 6.7% in WordPress (wiki).

A second decision point that developers encounter early on concerns the type of documentation to create. Contributors typically create one type of documentation initially and the documentation covers only a subset of the code. Then, as the project evolves, contributors create more documents of various kinds. After analyzing the interviews of both contributors and users, we identified three types of documentation based on their focus: a task is the unit of *getting started documentation* (Section 4.2), a programming language element (e.g., a function) is the unit of *reference documentation* (Section 4.3), and a concept is the unit of *conceptual documentation*. These documentation types are consistent with some previous classification attempts [2, 3].

Incremental Changes. Small and continuous incremental changes are the main force driving the evolution of open source project documentation. We noticed in our historical analysis that all changes except a few structural changes and the first revisions concerned a few words or a few lines of code and that these changes occurred regularly throughout the project history (see Table 4 in the Appendix). In this production mode, open source contributors encounter two major decision points: how to adapt the documentation to the project's evolution and how to manage the project community's contributions.

We found in our historical analysis that software evolution motivated at least 38% of the revisions to the documents we analyzed (adaptation and addition changes). We encountered five strategies (i.e., decisions) that contributors used to adapt the documentation to the project's evolution: contributors (1) updated the documentation with each change (Section 4.4), (2) updated the documentation before each release, (3) relied on a documentation team to document the changes (Section 4.5), (4) wrote the documentation before the change and used it as a specification, or (5) did not document their changes.

The second decision point contributors encounter is to determine how to manage the documentation contributions from the community. These contributions come in various forms: (1) documented code patches (Section 4.4), (2) documentation patches, (3) documentation hosted outside the official project's web site, (4) comments and questions asked on official support channels (Section 4.6), and (5) external support channels such as stackoverflow.com. Managing the documentation contributions represents a large fraction of the documentation effort: in our historical analysis, we found that 28% of the document revisions, excluding documents on wikis, originated from the community.

Bursts. During a project's lifetime, the documentation occasionally goes through major concerted changes that we call bursts. These changes improve the quality of the documentation, but they require such effort that they are not done regularly.

Publishers sometimes approach contributors of open source projects to write *books* about their projects: six contributors in our study mentioned that they (or their close collaborators) wrote books. One consequence of writing books is that contributors think more about their design decisions: *"it forced me to be more precise, to think carefully about what I wrote"*$_{C3}$.[8] This particular contributor made many small changes to clarify the content of the official documentation while he was writing the book. Because books about open source projects are not always updated, their main advantage lies in the improvement of the quality of the official documentation and the time that the contributors take to reflect on their design decisions.

Contributors also *change the documentation infrastructure* when it becomes too costly to maintain. Maintenance issues either come from custom tool chains, *"it is so complex that our release manager can't build the documentation on his machine"*$_{C4}$, or from a barrier of entry that is not high enough (e.g., wiki).

The last type of burst efforts are the major reviews initiated by the documentation contributors themselves. During these reviews, contributors can end up rewriting the whole documentation (C5) or simply restructuring its table of contents (C8). We observed that major reviews lasted from six weeks (C7) to three years (C9).

4.1 Wiki as Documentation Infrastructure

We begin our description of major decisions with the selection of a public wiki to host the documentation infrastructure. Wikis enable contributors to easily create a web site that allows anybody to contribute to the documentation, offers a simple editing syntax, and automatically keeps track of the changes to the documentation.

Context. Contributors select wikis to host their documentation when the programming language of the project is not associated with any infrastructure (such as CPAN with Perl) or when the project contributors want to rely on crowdsourcing to create documentation, i.e., they hope that users will create and manage the documentation.

Public wikis also offer one of the lowest barriers to entry: the contribution is one click away. According to contributors like C7, it is a powerful strategy to build a community around the project: C7 started to contribute on his project by fixing misspelled words.

Consequences. Although wikis initially appear to be an interesting choice for contributors, all the projects we surveyed that started on a wiki (4 out of 12) moved to an infrastructure where contributions to the documentation are more controlled. As one contributor mentioned: *"the quality of the contributions... it's been [hesitating] ok... sometimes [it] isn't factual so we had to change that... but the problem has been SPAM"*$_{C1}$. Indeed, we observed in our historical analysis that projects on wikis are often plagued by SPAM (24.1% of the revisions in Firefox) or by the addition of URLs that do not add any valuable content to the documentation (e.g., a link to a tutorial in a list already containing 20 links).

Another problem with wikis is that they lack authoritativeness, an important issue according to our users: *"I don't want to look at a wiki that might be outdated or incorrect"*$_{U3}$ For example, we observed cases such as a revision in a Firefox tutorial where one line of a code example was erroneously modified (possibly in good faith). The change was only discovered and reverted one day later (June 13th 2006).

Finally, because the barrier to entry is low, i.e., there is not much effort required to modify the documentation, the documentation can become less concise and focused over time: *"there's a user-driven desire to make sure that every single possible situation is addressed by the documentation. [these situations] were unhelpful at best and just clutter at worst"*$_{C5}$. According to C7, managing public wikis of large projects is a full-time job.

Alternatives. As users and contributors mentioned, the community is less inclined to contribute documentation than it is to contribute code, so the barrier to contribute documentation must be lower than the barrier to contribute code. Still, there exist mechanisms that encourage user contributions and that do not sacrifice authoritativeness, such as allowing user comments at the bottom of documents. Another strategy is to explicitly ask for feedback within the documents. For example, we observed in our historical analysis that Django provides a series of links to ask a question or to report an issue with the documentation on every page. Hibernate provides a similar link on the first page of the manual only. We could not find such a link in the Eclipse documentation. This strategy could explain in part the number of revisions that were motivated by the community: Django: 48%, Hibernate: 10%, and Eclipse: 10%.

4.2 Getting Started as Initial Documentation

Getting started documentation describes *how to use* a particular feature or a set of related features. It can range from a small code snippet (e.g., the synopsis section at the beginning of a Perl module) to a full scale tutorial (e.g., the four-part tutorial of Django).

Context. Contributors create getting started documentation as the first type of documentation so that users can install and try the project as quickly as possible. Contributor C8 mentioned that for open source projects, getting started documentation is the best kind of documentation to start with because once a user knows how to use the basic features, it is possible to look at the source code to learn the details of the API.

For seven contributors, "getting started" documentation has not only a training purpose, but it also serves as a marketing tool, it should *"hook users"*$_{C1}$, specifically when there are many projects competing in the same area. In contrast, the contributors of the five oldest projects reported that there was no marketing purpose behind the getting started documentation: these projects were the first to be released in their respective field and the contributors wrote the documentation for learning purpose only.

Contributors of libraries that offer atomic functions that do not interact with each other felt that getting started documentation was difficult to create because no reasonable code snippet could give an idea of the range of features offered by the libraries. These contributors still tried to create a document that listed the main features or the main differences with similar libraries.

Consequences. The importance of getting started documentation was confirmed by users who mentioned examples of projects they selected because their documentation enabled them to get started faster and to get a better idea

[8] Identifiers are associated with quotes for traceability and to distinguish between participants. Identifiers of contributors and users begin with a "C" and "U" respectively.

of the provided features. For example, U5 selected Django over Rails because the former had the best getting started documentation, even though the latter looked more *"powerful"*$_{U5}$. C2 confirmed that users evaluate Perl projects by looking at their synopsis.

Writing getting started documentation is challenging though: *"technical writing... I didn't have much exposure... I got used to it to some degree, but it is a challenge... it can take a lot of time"*$_{C6}$. Finding a good example on which to base the getting started documentation, an example that is realistic but not too contrived, is difficult (C11).

4.3 Reference Documentation as Initial Documentation

Contributors may decide to initially focus on reference documentation by systematically documenting the API, the properties, the options and the syntax used by a project.

Context. When a library offers mostly atomic functions, reference documentation is the most appropriate documentation type to begin with because, as contributor C11 mentioned, it can be difficult to create getting started documentation that shows examples calling many functions.

In contrast to libraries with atomic functions, frameworks expecting users to extend and use the framework in some specific ways need more than reference documentation according to the users we interviewed. Frameworks, by their nature, require users to compose many parts together, but reference documentation only focuses on one part at the time: *"interactions between these classes is often very difficult to get a grasp of... you need more information how the overall structure of the framework works"*$_{U4}$.

When contributors initially create the reference documentation, they either systematically document all parts of their projects or they rely on a more pragmatic approach: *"I try to go for anything that is not obvious"*$_{C10}$. Indeed, most programming languages are self-documenting when the types and members are clearly named. Users repeatedly confirmed that when a function has a clear name and a few well-named parameters, they will just try to call the function and will only seek the documentation if they encounter a problem. In statically typed languages such as Java, developers will use auto-complete to learn about the possible types and members and in dynamic languages such as Python, developers will execute code in an interpreter and call functions such as `help()` (displays API documentation) to learn more about a program. For weakly and statically typed languages such as C, developers cannot rely on type names (because many types are integer pointers) or on an facilities provided by an interpreter and reference documentation becomes more important. One user mentioned that the equivalent of an empty API documentation with only the type name, (e.g., Doxygen), could save him hours of source code exploration.

Consequences. Comprehensive reference documentation, especially for libraries, can contribute to the success of a project. For example, contributor C1 ensured that all functions of his project were documented before releasing the first version. According to C1, even if his project launched a year after a competing project, the user base grew quickly because the competing project had no documentation. Nowadays, although there are at least four other libraries providing similar features, C1's project has the largest user base and C1 attributes this success in large part to the documentation, a claim that was confirmed by four users.

In terms of effort, reference documentation is the easiest type of documentation to create according to the contributors we interviewed. For example, when C4 works with a developer who does not have strong technical writing skills, C4 works on the getting started documentation and let his colleague works on the reference documentation.

4.4 Documentation Update with Every Change

One strategy to adapt the documentation to a project's evolution is to document a change quickly after implementing it or requiring external developers to include documentation and tests with the code they contribute.

Context. Although all projects except two have a policy that all changes must be documented before a new version is released, we observed that seven contributors preferred to document their changes shortly after making them instead of waiting just before the release. These developers considered that documentation was part of their change task and they saw many benefits to this practice (described below).

Contributors C4, C8, and C10 ask external contributors to document their code contribution These three contributors want to ensure that the coverage of the code by the documentation stays constant and that the contribution is well thought-out. Contributors sometimes bend this policy to encourage more code contributions. For example, C10 accepts code contributions without documentation for his smaller projects or for experimental features in larger projects.

Consequences. Documenting changes as they are made ensures that all the *"user-accessible features are documented"*$_{C1}$, a documentation property that users often mentioned.

Another, perhaps more surprising, advantage of updating the documentation with every change is that it leads to a form of *"embarrassment-driven development [EDD]: when you have to demo something, and documentation is almost like having a demo, you'll fix it [usability issue] if it's really annoying"*$_{C10}$. Contributors reported that they modified their code while working on the documentation of their project to attempt to: (1) adopt a clearer terminology, (2) add new tools to reduce the time it takes to use the project, (3) improve the design of their project, or (4) improve the usability of the API.

We observed that EDD happens when contributors are working on getting started documentation and are describing how to accomplish common tasks: this is when contributors take the perspective of the users and must compose many parts of the technology together. EDD is possible when the development process exhibits these properties: contributors who write the documentation have code commit privileges, these contributors can modify the code without going through a lengthy approval process, and the documentation process is not totally separated from the coding process. For example, C4 mentioned that he tried to write documentation as soon as possible in the development process: *"it's common that I discover that when I'm writing [documentation] I need to change the design of the library because I discover that my design isn't explainable"*$_{C4}$.

Nevertheless, two contributors minimized the benefit of embarrassment-driven development. For example, in the project of C9, many contributors review each code change so most usability or design problems are caught during the review phase and not while writing documentation. Contributor C11 added that for libraries that provide atomic functions, unit tests covering the common scenarios will also

enable developers to detect API usability issues (e.g., is it easy to call this method in the unit test?).

One advantage of requiring code contribution to be documented is that it helps project maintainers to evaluate large contributions: *"I start with the documentation: if the documentation is good I have fairly good confidence in the implementation. It's pretty hard to have a well-documented system that is badly implemented"*$_{C4}$.

As users mentioned, one potential issue with requiring that all changes be documented is that developers might write content-free documentation: comprehensive policies established by C1 such as requiring a code example for each function help developers avoid this issue.

4.5 Use of a Separate Documentation Team

Three contributors, C5, C7, and C12, were part of a dedicated documentation team in their project.

Context. We observed that external contributors formed documentation teams and officially joined a project when the original code contributors believed that documentation was important to their project, but lacked motivation (i.e., they preferred to write code) or confidence in their documentation skills.

We observed two types of documentation teams. The first type (C5 and C7) is responsible for documenting everything, from the new features to in-depth tutorials. The other type (C12) is responsible for improving the documentation such as adding examples, polishing the writing style, or completing the documentation, but code contributors are still responsible for documenting their changes.

Consequences. Relying on a documentation team to document most changes (first type of team) had many disadvantages. First, code contributors outnumbered documentation contributors so the documentation lagged behind the implementation of new features, a situation that led to frustration, both for the users and the documentation team: *"our release cycle should include documentation itself, [we need] more than just API reference, [we need] prose that covers kind of usages things. We're releasing tons of code..., but there is no documentation for it and people got frustrated"*$_{C5}$. A second disadvantage was that developers who implemented the change did not become aware of usability issues on their own. All contributors who were in a documentation team mentioned that they sometimes acted as testers and reported issues with new features to the developers, but developers were not always receptive to their comments: *"somebody made a decision and it became that 'name', meanwhile someone in the documentation had made [another] decision on what the name would be... It was a very big struggle in naming things: [the developer name] confuses lots of things"*$_{C7}$. In contrast, C12, who is part of the second type of team, mentioned that the development team usually let the documentation team work on the terminology.

Contributors C7 and C12 mentioned that having a documentation team lowers the barrier to entry: users with no advanced knowledge of a programming language can still contribute to the documentation and become an official contributor with commit privileges.

4.6 Documentation Updates based on Questions

One strategy used by contributors to leverage the community is to consider questions asked on support channels (e.g., mailing list) to be a bug report on the documentation.

Context. The best example of this strategy came from Contributor C9 who sent us a list of emails that had been exchanged on the mailing list about an unclear section in the documentation. The exchange started with a question about the difference between two parameters: *"I see six emails... The problem is that the nuance of this particular command was really not clearly spelled out... This is a case where we really aren't doing our job"*$_{C9}$. C9 then attempted to edit the problematic section in the documentation and submitted a patch for review to the mailing list. After a few email exchanges with other contributors, C9 further edited the section and committed his changes. Overall, the change took at most 20 minutes. Other contributors described a similar experience when managing questions.

Consequences. Community feedback is essential to write clear documentation: *"when I write documentation, I skip things which need to be documented. But I am not aware of that. It's impossible to get around that problem unless you actually have someone else... who does not understand the details about the system"*$_{U4}$. In our historical analysis of changes, we found that more than half of the clarification changes (102 out of 195) were about explicitly stating something that was implied, such as adding an extra step to a tutorial. Community feedback helps locate these parts of the documentation that needs clarification and that could not have been foreseen by the contributors.

The main effort when continuously improving the documentation does not lie in the changes themselves but in constantly looking for occasions to improve the documentation. As C12 said, only a small percentage of the community contribute through the various channels (IRC, mailing list, bugs). A question raised by one individual on the mailing list might actually be asked by many more users. Many tricks are then used to evaluate if a question should be addressed by the documentation: was the question asked many times, is the answer provided by the community right or wrong, is the question addressed at all by the documentation, and is the question about an English-related issue and asked by a non-native English speaker?

Finally, users mentioned that they look for the presence of a live community when selecting a project: an active support channel gives some assurance to the users that their questions will be answered if they encounter any problem.

4.7 Summary

The decisions made by contributors have been presented as part of the documentation production modes and the decision points, which is useful to understand the context of these decisions. Yet, the consequences of the decisions for the contributors and the users are orthogonal to the documentation process. Table 3 shows the main consequence of the six decisions presented in Section 4 for five aspects of open source projects: the documentation creation effort, the documentation maintenance effort, the project adoption, the number and quality of community contributions, and the learnability of the technology. For each consequence, we indicate the number of contributors (C) or users (U) who discussed it. Because our questions and selection criteria evolved as the study progressed, these numbers reflect the variety of observations we gathered for each consequence: a quantitative study with a larger sample would form a natural step to evaluate the frequencies of these consequences.

Impact on − >	Doc. creation	Doc. maintenance	Adoption	Community contributions	Learnability
Public wiki	Made the creation faster (3C)	Increased maintenance (4C)	Very divergent opinions (4C,4U)	Lowered barrier to entry, led to low quality (4C)	Led to "corner cases clutter" (3C)
Getting started	Required strong writing skills (7C)		Was used as marketing tool (5C,3U)		Improved for framework (4C,6U)
Reference doc.	Was the easiest type to create (4C)	Was more costly to maintain (1C,1U)	Was mostly important for libraries & Competitive advantage. (1C,4U)		Improved for libraries (1C,8U)
Doc. update with changes	Required smaller upfront effort (2C)	Led to small but numerous changes more adapted to open source development process (5C)	Led to better coverage, a selection criteria (1C,5U)	Increased the barrier to entry but improved the quality of the contributions (3C)	EDD led to API usability improvement (4C)
Doc. team	Documentation lagged behind released features[10] (2C)	Documentation effort shifted from dev. team to doc. team (3C)		Lowered barrier to entry (2C)	Improved clarity and conciseness of documentation. (3C)
Updates based on questions	Lowered upfront effort (2C)	One of the main sources of maintenance (7C)	Was a sign of community activity, a selection criteria (2U)	Questions became a contribution (7C)	Led to many clarifications (7C)

Table 3: Decisions and their consequences

5. QUALITY AND CREDIBILITY

We evaluated the *quality* (are the findings innovative, thoughtful, useful?) and the *credibility* (are the findings trustworthy and do they reflect the participants', researchers', and readers' experiences with a phenomenon?) of our study by relying on three criteria proposed by Corbin and Strauss [5, p. 305]: fit, applicability, and sensitivity. These evaluation criteria are more relevant for a qualitative study than the usual threats to validity associated with quantitative studies [6, p.202]. The goal of our study was not to generalize a phenomenon observed in a sample to a population: instead we are generating a theory about a complex phenomenon from a set of observations obtained through theoretical sampling.

We produced a four-page summary presenting a subset of the decisions we analyzed and we invited the 12 contributors to review this summary to ensure that our findings resonated with their experience. Six contributors accepted our invitation and responded to a short questionnaire.

Fit. "Do the findings fit/resonate with the professionals for whom the research was intended and the participants?" The contributors found that the major decisions they made were represented in our summary. They mentioned though that many smaller decisions or factors were missing. For example, C7 remarked that the fact that documentation teams had to always catch up with the development team was not represented well in the summary. We had analyzed most of these details, but for the sake of brevity, we did not include them in the summary. There are only a few decisions that we did not analyze because we thought that they were less relevant to documentation (e.g., how to support users). These comments motivated our choice of presenting only a few important decisions and providing more detailed findings.

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" To the best of our knowledge, this is the first study on the process taken by contributors to create and maintain developer documentation. We hope that our description of the documentation decisions will help researchers devise documentation techniques that better support documentation decisions. For example, recognizing that the programming language plays an important role in the documentation decisions could lead to the development of solutions for languages that have a less standardized documentation culture.

We believe that this study has many benefits for practitioners. Contributor C11 mentioned that our summary could help other contributors reflect more on their documentation approach. Contributors and users mentioned that there is a general lack of motivation when it comes to contributing documentation. We hope that by uncovering the context and the consequences of documentation decisions, such as how documenting can improve the quality of the code, and how certain types of document contribute to the success of projects, could increase the motivation of contributors and users.

Sensitivity. "Were the questions driving the data collection arrived at through analysis, or were concepts and questions generated before the data were collected?" We did not enter this study with a blank slate because we have worked on documentation studies and tools in the past. To address this issue, Creswell recommends disclosure of any stance on the issue that researchers had before beginning the study [6, p.217]. For instance, we thought that writing documentation took a large amount of time and effort and we did not think that the community could play such a significant role in the documentation process. We were surprised at first to see the contributors struggle to name a single challenge to documentation. We soon realized how documentation could be seen as a vital and interesting part of open source projects and how the community could help improve the documentation. These early observations forced us to recognize and reconsider our preconceptions and helped us look at the data from a fresher perspective.

6. RELATED WORK

Most of the related work on developer documentation has focused on studying how developers use documentation and devising techniques to document programs.

[10]Only for documentation teams that are responsible for documenting changes

How Developers Learn Frameworks and Libraries. Kirk et al. conducted three case studies to study the problems encountered by software developers when using a framework [12]. They identified general kinds of questions such as finding out what features are provided by the framework and understanding how classes communicate together in the presence of inversion of control and subtle dependencies. The authors observed that different types of documentation provided answers to a subset of the questions.

Carroll et al. observed users reading documentation and found that the step-by-step progress induced by traditional documentation such as detailed tutorials and reference manuals was often interrupted by periods of self-initiated problem solving by users [3]. Indeed, users ignored steps and complete sections that did not seem related to real tasks, and they often made mistakes during their unsupervised exploration. Because this active way of learning was not what the designer of traditional documentation intended, Carroll et al. designed a new type of documentation, the minimal manual, that is task-oriented and that helps the users resolve errors [3, 4, 15].

Robillard conducted a survey and qualitative interviews in a study of how Microsoft developers learn APIs [14]. The study identified obstacles to API learnability in documentation such as the lack of code examples and the absence of task-oriented documentation. Forward and Lethbridge conducted a survey with developers and managers, and asked questions regarding the use and the characteristics or various software documents [11]. According to the participants, the following properties of software documentation were the most important: content (information in the document), up-to-dateness, availability, use of examples, and organization (sections, subsections, index).

Nykaza et al. performed a need assessment on the desired and required content of the documentation of a framework developed by a software organization [13]. The authors observed that junior programmers with deep knowledge of the domain and senior programmers with no knowledge of the domain had similar documentation needs about the framework. The programmers preferred simple code examples that they could copy and execute right away (as opposed to complex examples showing many features at once) and a manual that had self-contained sections so users could refer to it during their exploration (as opposed to manual that must be read from start to finish).

We complement these studies by studying the decisions made by the producers of documentation and by identifying the effort required by these decisions and their impact on the project and the users.

Documenting Programs. Many documentation techniques rely on mining code examples to infer usage information about libraries and frameworks. For example, SpotWeb mines code examples found on the web to recommend framework hotspots, i.e., classes and methods that are frequently reused [17]. MAPO mines open source repositories and indexes API usage patterns, i.e., sequence of method calls that are frequently invoked together [18]. Then, MAPO recommends code snippets that implement these patterns, based on the programming context of the user.

Schäfer et al. used a clustering technique to recover the main building blocks of a framework from client programs to build a representation of the framework that is easy to understand by users [16]. Similar classes are grouped together to help users understand the framework.

Finally, there are tools that help developers produce or augment existing documentation. Mismar is a semi-automated tool that produces tutorial-like wizards from concerns, i.e., the classes and methods related to a framework extension point, and that finds code examples that implement the tutorial [7]. eMoose highlights in a code editor the methods that have special rules described in their Javadoc [10].

We believe that tools can be useful to complement the documentation, but they cannot replace human-written documentation. As we observed in our study, some documents are used for marketing purposes so they cannot be generated, and writing documentation introduces a feedback loop that is beneficial for the program's usability.

7. CONCLUSION

Developers rely on documentation to learn how to use frameworks and libraries and to help them select the open source technologies that can fulfill their requirements. Following a qualitative study with 22 documentation contributors and users and the analysis of the evolution of 19 documents, we observed the decisions made by open source contributors in the context of three production modes: initial effort, incremental changes, and bursts.

Understanding how these decisions are made and what their consequences are can help researchers devise documentation techniques that are more suited to the documentation process of open source projects and that alleviate the issues we identified. Our findings can also help practitioners make more informed decisions. For example, a better understanding of embarrassment-driven development could motivate developers to document their changes quickly after making them. A better comprehension of the relationship between the type of project (e.g., library or framework) and getting started and reference documentation could help contributors focus their effort on the more appropriate type of documentation.

As a future work, we would like to report our results on the other decisions made by open source contributors and pursue our analysis of the documentation needs of users.

Acknowledgments

The authors thank Harold Ossher, Tristan Ratchford, Annie Ying and the anonymous reviewers for their valuable comments on the paper. This project was supported by NSERC.

8. REFERENCES

[1] S. Adolph, W. Hall, and P. Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *Proc. Conf. of the Center for Advanced Studies on Collaborative Research*, pages 166–178, 2008.

[2] G. Butler, P. Grogono, and F. Khendek. A reuse case perspective on documenting frameworks. In *Proc. IEEE Asia Pacific Soft. Eng. Conf.*, pages 94–101, 1998.

[3] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz. The minimal manual. *Journal of Human-Computer Interaction*, 3(2):123–153, 1987.

[4] I. Chai. *Framework Documentation: How to document object-oriented frameworks. An empirical study.* PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[5] J. Corbin and A. C. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory.* Sage Publications, 3rd edition, 2007.

Code	Django	WP	Plasma	Hib.	Spring	GTK	Firefox	DBI	Shoes	Eclipse	Avg.
Clarification	**18.2**	**7.1**	**13.1**	6.9	**14.1**	1.9	**9.5**	**16.6**	**5.9**	1.1	**9.4**
Adaptation	**15.8**	**8.7**	**24.6**	**17.2**	**8.8**	**33.3**	7.3	**32.4**	0	**18.5**	**16.7**
Addition	**16.7**	**19**	**18.0**	**24.1**	**20.9**	**9.3**	7.9	**27.6**	**64.7**	**5.4**	**21.4**
Structure	3.2	6.3	3.3	15.5	4.4	1.9	1.3	2.07	**8.8**	**6.5**	5.3
Format	**11.4**	6.7	3.3	3.5	6.6	**11.1**	3.2	1.4	**5.9**	**55.4**	**10.8**
Links	6.5	**27.7**	6.6	3.5	1.9	5.6	**14.2**	**4.8**	0	1.1	7.2
Correction	7.6	2.4	**11.5**	**15.5**	**10.6**	**13**	3.8	3.5	**5.9**	0.0	7.4
Polish	**20.5**	**17.4**	**13.1**	**13.8**	**32.8**	**18.5**	**11.7**	**11.7**	**5.9**	**10.9**	**15.6**
SPAM	0.0	2.4	1.6	0.0	0.0	0.0	**24.1**	0.0	0.0	0.0	2.8
Revert	0.0	2.4	4.9	0.0	0.0	5.6	**17.1**	0.0	2.9	1.1	3.4

Table 4: Classification of document revisions (in %). Top-5 codes for each document are in italic.

[6] J. W. Creswell. *Qualitative Inquiry and Research Design.* Sage Publications, 2nd edition, 2007.

[7] B. Dagenais and H. Ossher. Automatically locating framework extension examples. In *Proc. ACM SIGSOFT Intl Symposium on Foundations of Soft. Eng.*, pages 203–213, 2008.

[8] B. Dagenais, H. Ossher, R. K. Bellamy, M. P. Robillard, and J. P. de Vries. Moving into a new software project landscape. In *Proc. IEEE/ACM SIGSOFT Intl Conf. on Soft. Eng.*, pages 275–284, 2010.

[9] C. R. B. de Souza and D. F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proc. Intl Conf. on Soft. Eng.*, pages 241–250, 2008.

[10] U. Dekel and J. D. Herbsleb. Improving api documentation usability with knowledge pushing. In *Proc. IEEE/ACM SIGSOFT Intl Conf. on Soft. Eng.*, pages 320–330, 2009.

[11] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proc. ACM Symposium on Document Engineering*, pages 26–33, 2002.

[12] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in object-oriented framework reuse. *Journal of Empirical Soft. Eng.*, 12(3):243–274, 2007.

[13] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proc. Intl Conf. on Computer Documentation*, pages 133–141, 2002.

[14] M. P. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[15] M. B. Rosson, J. M. Carrol, and R. K. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proc. ACM SIGCHI Conf. on Human Factors in Computing Systems*, pages 423–430, 1990.

[16] T. Schäfer, I. Aracic, M. Merz, M. Mezini, and K. Ostermann. Clustering for generating framework top-level views. In *Proc. Working Conf. on Reverse Eng.*, pages 239–248, 2007.

[17] S. Thummalapenta and T. Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proc. IEEE/ACM Intl Conf. on Automated Soft. Eng.*, pages 327–336, 2008.

[18] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. European Conf. on Object-Oriented Programming*, pages 318–343, 2009.

Appendix A. Results of Historical Analysis

We classified each documentation revision by associating a category summarizing the rationale behind the change. When there were multiple types of change, we identified the change that had caused the largest number of lines in the document to be modified. Ten categories of change emerged from our analysis:

Clarification. Addition of a note or the modification of words to clarify existing content.

Adaptation. Modification of the text to reflect the new state of the project. An adaptive change can range from the update of copyright date to the recommendation of a new feature over and old one.

Documentation Tools and Infrastructures	
CPAN	`www.cpan.org`
POD	`perldoc.perl.org/perlpod.html`
Sphinx	`sphinx.pocoo.org`
Javadoc	`java.sun.com/j2se/javadoc`
Doxygen	`www.doxygen.org`
Projects	
Django	`www.djangoproject.com`
WordPress	`wordpress.org`
KDE Plasma	`plasma.kde.org`
Hibernate	`www.hibernate.org`
Spring	`www.springsource.org`
GTK+	`www.gtk.org`
Firefox	`www.mozilla.com/firefox`
DBI	`dbi.perl.org`
Shoes	`github.com/shoes/shoes`
Eclipse	`www.eclipse.org`
Rails	`rubyonrails.org`

Table 5: Documentation tools and open dource projects mentioned in this paper

Addition. Text or examples that are added to a document. For example, when a new feature is released, a section describing the feature is often added in a reference manual.

Structure. When sections are moved inside or outside documents, e.g., when a large document is split in multiple smaller documents.

Format. Modifications of the file syntax, e.g., the addition of an HTML closing tag that had been forgotten in the previous revision.

Links. Addition of a URL to the documentation.

Correction. Modification of a code example because it was broken or the behavior was not the one intended. Because it was not always possible to determine if a correction was due to refactoring, the modifications of a code example following a refactoring are included in this category.

Polish. Words or sentences that are copy edited, e.g., spelling error. When new sentences or domain-specific words were added to clarify an existing sentence, we considered the change to be part of the clarification category.

SPAM. Unsolicited advertisement or vandalism.

Revert. When the current version of the document is reverted to a previous version. This is often caused by SPAM, but incorrect or unclear addition by contributors can also cause a revert.

Table 4 shows the distribution of the change categories across the document revisions for each project. The five most popular categories in each project are in bold. The last column, Avg., presents an unweighted average of each category across the 10 projects: because we did not analyze the same number of documents and revisions for each project, a weighted average would be heavily biased toward the documents with the most revisions. We found that the top five category in weighted and unweighted averages were the same.