Open-Source Documentation: In Search of User-Driven, Just-in-Time Writing

Erik Berglund
Linköping University
S-581 83, Linköping,
Sweden
+ 46 13 28 24 93
eribe@ida.liu.se

Michael Priestley
IBM Toronto Lab
Canada
mpriestl@ca.ibm.com

ABSTRACT

Iterative development models allow developers to respond quickly to changing user requirements, but place increasing demands on writers who must handle increasing amounts of change with ever-decreasing resources. In the software development world, one solution to this problem is open-source development: allowing the users to set requirements and priorities by actually contributing to the development of the software. This results in just-in-time software improvements that are explicitly user-driven, since they are actually developed by users.

In this article we will discuss how the open source model can be extended to the development of documentation. In many open-source projects, the role of writer has remained unchanged: documentation development remains a specialized activity, owned by a single writer or group of writers, who work as best they can with key developers and frequently out-of-date specification documents. However, a potentially more rewarding approach is to open the development of the documentation to the same sort of community involvement that gives rise to the software: using forums and mailing lists as the tools for developing documentation, driven by debate and dialogue among the actual users and developers.

Just as open-source development blurs the line between user and developer, open-source documentation will blur the line between reader and writer. Someone who is a novice reader in one area may be an expert author in another. Two key activities emerge for the technical writer in such a model: as gatekeeper and moderator for FAQs and formal documentation, and as literate expert user of the system they are documenting.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**] User Interfaces – *Training, help, and documentation*.

General Terms

Documentation, Human Factors

Keywords

Open source documentation, just-in-time, user-driven.

1. THE PROBLEM

Over the years, the software industry has accepted that changing requirements are simply part of the software development process. An allowance for client requirements change, even an expectation of change, is at the foundation of most software development methodologies. The Rational Unified Process (RUP) illustrates this, and Extreme Programming (XP) exemplifies it. Taken to the extreme, as it often is in open-source development, the functionality of the product may not be determined until the day it is completed.

Continuous requirements change makes traditional methods of software documentation difficult. Measured from the last change, production lead-time is effectively nil. While some projects do incorporate documentation requirements into their production schedule, in many cases writers simply have to make the best of an impossible situation, and produce what documentation they can under the circumstances.

Writers cannot simply adhere to a pre-existing plan: they have to quickly assess the relevance of each change and assign priorities to each affected area. Throwing more writers at the problem is a solution with a rapidly diminishing return on investment: more writers typically require more coordination and planning, not less, and this compounds the risks posed by a volatile information domain. The problem cannot be solved with more planning or more reviewing. The writer simply has to make the most of what resources are available, and aim to produce something useful at the end of it.

Applying software development methods to the writing process may sound like a plausible solution to the problem [36]. However, the solution falls short when documentation departments lack the resources and influence that would allow them to negotiate changes after the manner of development departments. While process, and especially integration of process [25], can help writers track changes, it doesn't help them find the resources or time to make changes. Application of processes and integration of processes provide only half the answer: they provide knowledge, but not the opportunity to apply it.

So the problem, finally, is that when we have the understanding, we have it too late; and regardless of how well we plan or how hard we work, the best we can hope for is an incomplete manual and help set that have a minimum of errors.

2. THE SOLUTION

There are various ways to address this problem, innovations in how we write (in small reusable units), how we process (using various singlesourcing technologies), and how we ship to the customer (incrementally over the web, through a knowledge base, and so on and so forth). These solutions are useful, and make the most of what resources are available.

But a bolder solution is to simply accept that what we are shipping is incomplete, that documentation is in fact inherently incomplete, and then move on to the larger problem: how can we provide our customers with the answers to their questions?

Software documentation has been trending to the minimalist for quite some time. As software becomes more usable, it often picks up document-like attributes (from GUIs to embedded text to wizards), and becomes to some extent self-documenting, lifting some of the burden of completeness from the documentation. There's no need to document the obvious: when the software is self-explanatory (would that it were more often), the documentation can afford to be mute.

Unfortunately, as the same explanation will not serve all users, the same piece of software may be self-explanatory for some and completely opaque to others. This would seem to put the burden of completeness back onto documentation: even if a feature is obvious for one user it isn't for all, therefore document all features. While this conclusion is valid enough when we consider documentation as a static, published entity (something produced with the product for the product), the situation becomes more complex when we think of documentation as a networked and evolving entity, a larger world of information resources in which static documentation provides only a starting point.

In other words, shipping incomplete documentation may be acceptable if the information gaps can be filled in some other way, after the shipping date, as the answers become needed. This is a step beyond print-on-demand, to write-on-demand. Such user-driven, just-in-time production of content would also strengthen relevance in content production and foster communities building on a global scale.

How would write-on-demand processes work? User-driven, just-in-time documentation depends first on the availability of a community of users who can request and receive documentation. You cannot provide the answers without the ability to hear the questions. Users may be prepared to wait for an answer, if they know one is forthcoming. Further, a user may be prepared to collaborate in the answer, providing parts they do know if only to help speed up the writer's research time. In fact, herein lies the heart of our solution. The burden of completeness is derived from the fact that different users require explanations of different features: obviousness is subjective. But this same fact in a networked world implies the opposite: for every user who is confused by a feature, there is another user who understands it and can explain it. The corollary of partial confusion is partial understanding. The users themselves can fill in the holes. In fact, this is how mailing lists and discussion forums work. The role of the writer, in a situation like this, is to be in effect a sort of super-user: someone who is articulate and knowledgeable and regularly available to the community.

In software development, there is already a methodology that is based on such processes: open source development. In recent years, the open-source approach to software development has resulted in notable success stories: Linux [12], Mozilla or Netscape [13], and the Apache web server (over 50% of the market) [1, 15] are all large, global products in fast moving technical areas. Open-source development, in its purest form, is an ecological process with a focus on user-driven just-in-time production of content. The community develops what it needs when it needs it bad enough. Software grows from the needs, desires, and work of the community. Given the success of open-source development as a response to these problems in software development, it may be worth considering how the same methodologies can be applied to software documentation.

2.1 Open-source documentation and technical writing

In this paper we will discuss open-source documentation as a user-driven, just-in-time documentation process that delivers the documentation users want when they want it. In a sense, open-source development of documentation is practiced continuously today. Evolving content in mailing lists and FAQs are both the result and fodder for ongoing discussions that help develop a community's understanding of software products. Mailing lists and FAQs represent *technical debate* in user communities, which both answer questions about products and also discuss future development of products.

This paper addresses how technical debate can be turned into formal support for software products. We present an open-source documentation method focusing on debate and dialogue as the engines of content creation. Content extraction and debate moderation are also regarded as means for directing and transforming the tacit knowledge of the group into the explicit support for a technology. We will also address contemporary technical writing techniques in relation to the vision of open-source documentation, and discuss the changes that open-source documentation processes may bring about for the writing profession.

2.2 Organization

The paper is organized as follows. Section 3 analyses and describes open source development from the experiences of open-source software development. It also describes why open source development results in just-in-time, user-driven production of content. Section 4 provides a framework for open-source documentation projects and discusses how to achieve documentation through user contributions. Section 5 examines writing techniques in search for open-source processes. Section 6 discusses the state of the profession on open-source documentation projects. Finally, Section 7 summarizes the paper and discusses whether open-source documentation would work.

3. OPEN-SOURCE DEVELOPMENT

Open-source projects have received a fair bit of attention in recent years, with successful projects such as the Linux [12] operating system, the Apache web server [1], the Mozilla web browser [13], and the Perl and Python programming languages [22, 26]. According to the open-source initiative (OSI), a non-profit corporation dedicated to managing and promoting an open-source definition:

> The basic idea behind open source is very simple. When programmers can read, redistribute, and modify the source code for a piece of software, the software evolves. People improve it, people adapt it, people fix bugs. And this can happen at a speed that, if one is used to the slow pace of conventional software development, seems astonishing.
>
> – Open Source Initiatie web site [18]

Many open-source projects are developed as freeware but this is not a necessity of open-source projects. Though open source has its roots in freeware initiatives such as the GNU projects [9] of which the Emacs editor [7] is the most famous application [33], open source does not necessarily mean non-profit.

> Teaching new users about freedom became more difficult in 1998, when a part of the community decided to stop using the term "free software" and say "open-source software" instead.
>
> "Free software" and "Open Source" describe the same category of software, more or less, but say different things about the software, and about values. The GNU Project continues to use the term "free software," to express the idea that freedom, not just technology, is important.
>
> – Stallman R. [33]

The OSI definition of open-source does not exclude sales of open-source products, in fact it specifically mentions sales. It is the control over the source code that is key to the open-source certification that OSI provides [19].

The OSI certificate protects the source code's ability to move freely though different development projects. This gives the potential for a critical-mass effect, in which the efforts of many globally distributed independent groups with different goals jointly develop software that is more powerful than anything they could have developed individually. In a sense, the software becomes a completely independent entity, which can grow and evolve in directions its original developers never envisioned. According to Bruce Perens, who wrote the original draft of the OSI definition for the Debian open-source project [3], the definition is a bill of rights for the computer user. Certain rights are required in software licenses for that software to be certified as Open Source [20]. Essentially the right to:

- Make copies of the program, and distribute those copies
- Have access to the software's source code, a necessary preliminary before you can change it
- Make improvements to the program

While this bill of rights adequately defines when software is open source (and amenable to open-source development), it does not really describe the nature of open source development. For instance, the famous open-source projects such as Linux, Mozilla and Apache have had large and organizationally independent groups contributing to the same development. How such groups can cooperate, and how a community with a range of involvement from individuals to companies can organize itself, are aspects that are not covered by the OSI definition.

Open source is often described as massive parallel development [5, 27, 28, 30]. Furthermore, open source is often connected with individuals working together in a highly decentralized organization. The primary technological drivers for open source software include the need for more robust code, faster development cycles, higher standards of quality, reliability and stability, and more open standards/platforms. [5] Robustness is also one of the established benefits of open source [39, 21]. Perkins writes that it is, in fact, the decentralized organization that helps the open-source community to consistently produce powerful, robust, useful software solutions [21]. From a research perspective, open-source is a new but relevant area of investigation. The 1:a workshop on open-source-software engineering was held at the international conference on software engineering (ICSE) 2001, which hopefully will result in more research on the subject [6]. One of the few in-depth analysis of open–source can be found in Feller and Fitzgerald's framework analysis of open source software development [5]. Furthermore, the book *Open Sources: Voices from the Open-Source Revolution* provide articles written by key figures in the early days of open source [4].

The nature of open-source development still remains somewhat uncharted territory but is typically (among other characteristics) robust, public, just-in-time, user-driven, global, community-oriented, critical-mass dependent, non-directional in its growth, developed from the bottom up, and change-prone. We will elaborate on two aspects of open-source development: *user-driven* and *just-in-time*. The strength of these aspects is the focus they naturally put on relevance and priority. What gets built is what the users want when they want it bad enough.

3.1 User-driven

In many cases, open-source development is driven by demand for the product in the programming community itself [37]. Users develop the systems they need or want themselves. As such, open-source development can be viewed as an ecological process, in which independent users jointly grow their desired systems. In this its purest form, open-source users are open-source developers. This approach makes the most sense for projects that are relevant to large groups of people, because small groups cannot generate the hours to develop a major system. The basis for open-source development is massive parallel development. [27, 5] Also, open-source projects can be utterly decentralized where no authority dictates what who shall work on and how. Still tremendous organization and cooperation emerges. [Perkins 1999]

Of course, to grow substantially from the efforts of a user-community an open-source project must generate a critical mass of developers that contribute. This is what successful projects, such as Linux and Apache, have done. Also, the critical mass of users must be competent enough to understand and contribute on a highly detailed level – for instance system administrators – and as a result their needs will shine through in the software they produces. Which explains why, in the past, opens source projects mostly have been focused on operating and networking software, utilities, development tools, and infrastructure components [5].

Of course, for products such as Linux the majority of users will, if the projects is successful, eventually be users in the traditional sense that do not add to the functionality of the code or even have the ability or intent to contribute. However, the open communication channels used in open-source communication

(mailing list and web sites) still broadcast information and discussions to the world. Development is open also to those not directly involved and they may participate to lobby for functionality they need.

3.2 Just-In-Time

Open-source development can be considered just-in-time development because the users develop what they want when they want it bad enough. Of course, skeptics may argue that open-source development is mostly technically driven (and support technical desires rather than user needs) because people with technical skill define requirements by implementing them. However, in many open-source projects where the users are in fact technical people (for example, Perl and Apache) these distinctions become meaningless: technical desires are, in fact, the user needs.

Open-source projects are defined by very short release cycles [5]. According to Eric S. Raymond, one of the smart things Linus Torvaldsson did was to create an extremely short release cycle. Linus succeeded in getting solid feedback and responding to it in only 24 hours, something thought utterly bizarre at the time [29]. In this sense, Linus was also sensitive to requests in a just-in-time fashion and provided his community with rapid responses to their interest in the Linux project. So even when users are not implementing features themselves, the short cycle times and community involvement that typify open-source projects still provide just-in-time development.

4. AN OPEN-SOURCE DOCUMENTATION FRAMEWORK

Just as open-source development requires a framework through which a community can cooperatively develop code, open-source documentation requires a framework that captures the relevant qualities of open-source development (just-in-time and user-driven development) while accommodating the special requirements of documentation development.

The first step is simply to allow people to contribute, as Jones pointed out in a short article on open source and digital libraries [10]. Writing cannot be restricted to a privileged few: people outside the organization must be allowed to contribute. This is actually easier to consider for documentation, given that documentation is less dangerous in its possible effects (a badly written document won't erase your hard-drive - at least not directly - in the way software can).

The goal of the framework is to turn technical debate, currently taking place in mailing lists and discussion forums, into formal support for software products. In this section we define an open-source framework which is in subsequent sections matched with contemporary forums for technical debate and current technical writing techniques.

Open-source documentation should perhaps not be seen as text created through an open-source development model but rather as drawing from an accumulated pool of resources, which include both captured competence (text, multimedia) and living (persons) competence. An open-source framework can encourage the creation of these resources, from which a *documentation build* (by analogy to code builds) can create tutorials, standard documents, books, online reference manuals, and so forth as necessary for a particular project or delivery context.

4.1 Premises

There are a number of premises that must be met to even start considering open source documentation:

4.1.1 Electronic Documentation

An absolute requirement for open-source documentation is the electronic format. Open source projects must be editable on a global scale and it therefore becomes practically impossible to use print. However, this does not mean that the layout should exclude printable versions of the documentation because users will still want to print documentation. Hard-copy versions may, of course, be constructed from documentation builds.

4.1.2 Web-Site Driven

Since documentation source needs to be accessible to a global community of users, web sites are the logical organization and access mechanism. The easiest way to get started is to run your web site on a SourceForge server (either on the international SourceForge server at www.sourceforge.com or on your own downloaded copy of it) [32]. This provides a good starting point for managing your source via the web.

4.1.3 Open-source documentation License

Letting go of control requires the definition of a license over ownership of the open-source documentation and the ability to freely use the documentation source in documentation builds. The documentation source must be free to become part of many different projects. This includes allowing others to make documentation builds from the documentation source and even create new products from that pool. Without an open-source documentation license, there is less incentive for diverse groups to contribute to the effort, and little chance of achieving the necessary critical mass of contributors. Explicit open-source documentation licenses are also needed because the copyright applies to work regardless of medium and without copyright notice. For a discussion on copyright see the Stanford Copyright and Fair Use web site [34]

Open-source documentation license do exist today, among which GNU Free Documentation License [8] and the Open Content License [17] are the most commonly used for open source projects. These licenses allow distributions of verbatim copies and derived work under certain conditions. For instance, the GNU Free Documentation License allows distribution as long as the distributed copy also use the same license.

> We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does.
>
> – GNU Free Documentation License [8]

4.1.4 Documentation Splits

Along with the open-source documentation license comes an acceptance of the possibility of branching projects. This allows fundamental disagreements in a community to be resolved through splitting the community and creating a new version that is maintained in parallel with the original.

4.1.5 First Prototype

The open-source documentation project must start with a small first prototype that jump-starts the process and makes it believable that the project will result in something valuable and worthwhile. This first prototype, and the first documentation build, are, in practice, the sales pitch for the project. The prototype must not be complete but rather make it believable that a relevant result can be produced. Subsequent documentation builds do not have to match the vision of the first prototype: its purpose is to provide a departure point, not an end-point for development.

4.2 On-Going Support

Once the project gets started there are a number of aspects that need special attention to keep the project running smoothly:

4.2.1 User Control

As in all projects, the quality of the content needs to be regulated. Control is a social issue in open source development, in which the community regulates itself [10]. Typically the community grants certain serious and dedicated users special rights that allow them to review contributions and disallow illegal or inappropriate submissions. Naturally the writing staff will be among such power users but people outside the organization must also be allowed to regulate content.

4.2.2 Social Structure

User control requires the construction of a social structure for the members of the community. Assignment of power-user status can be based on engagement, seniority and peer ranking. Many open source systems, such as Source Forge, use peer ranking. The Source Forge ranking system measures teamwork/attitude, coding ability, design/architectural ability, follow-through/reliability, and leadership/management. Social ranking has other advantages as well. For instance, social ranking acts as recognition of contribution and as rewards. Furthermore, social ranking organizes users in relation to their capacity and therefore also organizes users into resources. Social structures also support the feeling of a community.

In a documentation project, coding could simply be replaced with writing as a ranked competency. However, in a mixed project (where the documentation is being developed alongside a particular piece of software), it would be worthwhile to define separate measurements for writing and for information design/architecture, to allow meaningful rankings of good developers who are poor writers and vice versa.

4.3 Goals

4.3.1 Building Documentation

The focus of the open-source documentation project should be to build documentation of more traditional style, such as user guides and how-to documents. The documentation source should not be regarded as documentation in itself. There is a risk in open-source documentation that web-based information repositories similar to article collections replace documentation. Such repositories are likely to spread information around and make reading difficult by requiring the reader to perform extensive search and content extraction.

4.3.2 Short Release Cycles

Documentation should have short release cycles to accommodate the flow of requirements and implementations, such as questions and answers. Short cycles are not just good service, it is a necessity for the continuous accumulation of content. Short release cycles is another requirement for user-driven process because large development resources are required. Such a design will require the constant build of documentation from the documentation source perhaps even every 24 hours. In this sense, letting go of control is essential because the task of gate keeping a large documentation source within 24 hours requires manpower and trust. A power-user social structure helps appoint trustworthy users that can change with little or no intervention.

4.3.3 Live Communication Forums

An aspect of documentation creation that differs from code creation is that live, people-to-people communication can become an integral part of the process. Chats with power-users, people who are particularly knowledgeable, can be held and recorded as part of the actual documentation-source. Web-cams can also be utilized to provide live feedback that can also be collected and stored. Such live content transmissions also help build the sense of a community.

4.3.4 Automatic Correctness Verification

In open source software projects, a compiler is often used to verify that only syntactically correct programming is added to the common resource pool. Code that does not pass compilation is not accepted. Beyond compilation, verification is provided through the massive parallel development inherent in open source. Similarly, open source documentation projects could have a number of automatic checks on content, including DTD validation for XML or SGML source, HTMLTidy reports for HTML and XHTML, spellchecks, linkchecks, and so forth.

4.3.5 Writing by Moderating

The technical writing staff responsible for the open source project should take care of moving content around, improving language, correcting errors, identifying gaps, and so forth rather than concentrating solely on writing the content. This staff must also write the first documentation prototype.

4.3.6 Discussion through Annotation

Discussion forums and mailing lists are typically organized chronologically and by subject (“threads”). Documentation, however, needs to be based on topics or tasks, organized into FAQ documents. The transformation from chronological and thread-based organization to more architected FAQs, and the rechunking from threads to topics and tasks, is a core concern of the documentation project.

Traditionally this has been done by hand, through either cut and paste or more complete rewriting. A more dynamic solution might be add metadata to the threads, allowing for more intelligent searching of archived discussions. However, this approach only allows for search-based exploration of relevant topics, and requires constant updating of the metadata. A more integrated solution would be to directly annotate the text in each message, calling out explicitly what part of it is a query and what part is an answer. Query lists can, naturally, be generated from the source for users talented enough to answer them for the writing staff. Answers that have already been provided by the community can be assessed according to the ranked skill of the author, and edited if necessary by posting the edited answer to the end of the thread.

Discussion through annotation naturally adds user comments and discussion to a topic framework, unlike thread-based discussion, which require transformation. In this sense, annotation speeds up content-extraction process and thereby shortens the release cycles for documentation builds.

4.3.7 Multiple Views

A big part of documentation is navigation and as the documentation source grows the navigation problem grows. Navigation is also personal or task dependent and it is therefore difficult to generate a general but effective index. Multiple indices, however, can exist and this may well be one of the larger sections of an open source documentation project. By allowing the construction of navigational links across documentation based on user design the navigation infrastructure can evolve and grow with time.

4.4 Technical Questions

There are a number of technical issues that need to be addressed by the open source documentation framework:

4.4.1 Documentation Format

The web infrastructure and the openness make the technical issue difficult. The need for a web-site driven project, the formats usable become somewhat limited. For annotation systems (i.e., direct additions to the documentation source) the system must work directly in the browser. This requirement makes XML a highly relevant documentation format because the basic web infrastructure supports XML. However, automatic spell correction needs to be present as well which may make things a bit more difficult today. For longer comments, individuals can be free to use whatever word processor they like to construct their answers as long as they can convert to the project format.

4.4.2 Documentation Layout and Author Reliability

The layout of a documentation system that includes questions and answers from the user community needs to show the reliability of content. At least, the system should clearly indicate that the source is open for contribution from a worldwide community allowing participation from, in principle, anyone with web access. The annotated manual for the PHP open source project does this in two ways: by calling the manual annotated and by displaying annotations from users in differently styled sections of the text [23]. Readers need to be made aware of who the writer is and their degree of competence.

4.5 Lifecycle

Initially, a documentation prototype provides the starting point for contributions from an open-source documentation community. As the project progresses, more and more of the content may be derived directly from the community, following a process of content creation and documentation builds can be summarized by the following lifecycle:

1. A user asks a question, either about existing content or by requesting information. The question is added to the source as a comment or as a new question.
2. Another user (may be a member of the writing staff) finds the question in some build from the source, perhaps a query listing or as part of a documentation build. The user answers the question and the answer is added to the source.
3. Other users provide answers, confirms answers or, adds comments and reposts to the source as an annotation.
4. Another user with editorial skills reworks the answer to and reposts.
5. The answer is automatically picked up in the next FAQ build, although ranked fairly low since it has only been asked once. The build may also validate that the FAQ has been correctly authored as a task, has no spelling errors, etc.
6. Another user with information architecture skills adds a reference to the task to pull it into the appropriate place in the overall task flow, and to include it in the appropriate indexes and tables of contents for whichever delivery contexts are appropriate.
7. Someone reads the documentation, has a problem with it, and asks a new question.
8. Repeat until software and documentation are perfect or obsolete, whichever comes first.

Alongside this process, documentation builds are continuously created from the source with layout visualizing the credibility of the different pieces. As a question-answer cycle matures the content become more and more integrated in the documentation by shifting style and location in the builds. Peers rate contributors that increase the status of such users. Automatic rating systems can be built in to the discussion format by measuring the addition of agreement, refinement, or disagreement to answers. For highly rated users, the technical staff investigates whether or not to grant user more privileges to cut corners in the gate keeping process.

4.6 Summary of Framework

The open-source documentation presented in this section focus on the creation of a user community that builds documentation by debating topics in a documentation source. From the source, documentation is built by extraction (automated if possible). The layout visualizes the credibility of content in style and position. As content mature through the community process, its visibility in subsequent documentation build releases increase.

Compared to traditional writing, open-source documentation focus on the user-driven, just-in-time aspects of content creation and the natural focus they put on relevance and priority.

5. OPEN WRITING TECHNIQUES

Open-source documentation also requires writing techniques that support the process of user-driven, just-in-time construction of documentation through an open-source model. In this section we discuss what current writing techniques offer in this respect.

5.1 Writing Reusable Units

Many online documentation projects currently use topic-oriented writing and information typing as ways to produce disciplined reusable information. Combined with task-oriented minimalism [2], these techniques can result in highly focussed, reusable, and user-focussed documentation. The question is how much of these techniques can be made accessible to a wide community, and can how consistency and accuracy be maintained, outside of the standard edit-publish-review cycle?

While various architectures define a variety of sizes and types of information, at minimum an information-typing architecture

defines the size of a topic (a single reusable "chunk" that describes a single idea, task, or thing) and three information types: concept, task, and reference. Multiple topics can be combined into task flows, organized by index or table of contents, and aggregated into books or websites [24].

Topic-oriented writing can seem quite alien to an accomplished technical writer more familiar with books, and there is often a significant learning curve associated with the change in writing goals and style. However, different as they are from a manual, they are in fact quite a natural fit for derivation from FAQs. Different types of question conform quite naturally to information types: how-do-I questions have tasks as answers, what-is-a or how-does-it-work questions have concepts or reference topics as answers. In addition, with the exception of extraordinarily long or vague questions, most FAQs are going to be naturally chunked at about the right size for a topic.

So is the fit between newsgroup source and topic-oriented, reusable content as easy as the normal gathering process that gives us FAQs? Nearly. Typing and chunking are the two main goals of an information typing architecture, but a website or book constructed out of topics needs coherence in its style and structure to look more than merely accidental, and to be predictable enough to be useful and usable.

5.2 Editors and Architects

The task of enforcing structural and stylistic guidelines can be in part taken up by the social structure: appointed or elected editors (users or contributors with highly rated writing and information architecture skills) can be reviewers and approvers of candidate topics. For example, in the case of topics harvested directly from marked-up newsgroup posts (as described in section 4.3.5), an editor could be required to forward the (edited, annotated) answer back to the group before it was considered a candidate for harvesting. Otherwise, contributors with editorial approval could perform the harvesting themselves, and impose a certain level of consistency as they went.

The two proposed skill measurements - writing and information architecture - point to two separate roles: the topic-level editor, who pays more attention to style and low-level content issues, and the collection-level editor, who defines the task flows and tables of contents that organize the topics into useful collections.

These two roles, and their responsibilities in a more structured development process, have been described in detail in [25].

5.3 Enforcing Structure with Markup

Structural guidelines can also be enforced by the use of a specialized markup language, whose DTDs or schemas prescribe particular structures for particular kinds of information. There are several possibilities for enforcing such structures:

5.3.1 HTML or XHTML

HTML is a very general standard, and as a result it does not usefully constrain the information you write in it: two equally valid topics (according to the HTML standard) can be as different as any two pages on the web. This is still better than complete chaos, however, and tools such as HTMLTidy make it easy to eliminate tagging errors. XHTML is somewhat better, and has the two advantages of being customizable (you can choose which modules you require) and, as part of the XML universe, addressable with XSLT and XPath, which makes it easy to transform and reuse.

5.3.2 DocBook

DocBook is a more specific standard than HTML, and out of the box it is focussed on book authoring. While DocBook provides better validation than HTML or XHTML, and has a good set of output transforms and tools, it is not particular topic-oriented. However, parts of it are highly structured, and could be used for specific domains (such as messages) as-is.

5.3.3 Customized DocBook

Generally speaking, if you want to use DocBook, you will need to customize it. This is a well-documented process, with the warning that if you want to add your own tags (not just choose a subset of the DocBook ones) you'll need to write your own transforms and tools.

5.3.4 DITA

The Darwin Information Typing Architecture is a topic-oriented information typing architecture for writing and publishing technical documentation. Out of the box, it is oriented towards creating information-typed topics (concepts, tasks, and reference), and is quite restrictive in its structures (especially for tasks). However, it is a new and still-evolving architecture, and there are a limited number of transforms available (PDF via FO and HTML are available outputs at the time this paper was written).

5.3.5 Specialized DITA

The good news is that you can create specialized topic types (such as EJB API descriptions, configuration file formats, cooperative tasks, etc.) quickly and easily. Generally speaking, the more closely you tailor your topic's structures and tags to your domain (the particular kind of software you are documenting, for example) the easier it will be to learn (because it matches what the writers are trying to create) and the more it can enforce structural consistency. The more tightly you scope your domain, the more exactly you can define your content rules, and the more precisely you can control consistency, before an editor even gets involved.

5.4 Massive Parallel Writing

Topic-oriented chunks written by users, refined by editors and architects, and confined by markup languages can help get contributions right from the start. Users can acquire the writing skills to a certain degree and the ones that learn the most also get the highest ranking and the social structure thereby help produce quality documentation. At some point, however, technique, editors, architects, and markup may not be enough. This is where one of the fundamental points of open-source development kicks in – massive parallel writing. When writers can read, redistribute, and modify the documentation source, the documentation evolves and become robust. People improve it, people adapt it, people fix bugs (see Section 3). If writing technique fails, open-source documentation will rely on the sheer size of a committed user community.

6. CONTEMPORARY OPEN-SOURCE DOCUMENTATION

Though genuinely open-source documentation cannot always be found even among open-source software projects, there are some

documentation projects and communication media that contain the user-driven, just-in-time production aspects we are searching for. Discussion forums, mailing lists, online annotated manuals, online editable manuals, and open-source documentation projects can be considered user-driven and just-in-time, but they do not necessarily conform to other aspects of our framework.

For instance, even when documentation uses an electronic format and is web accessible, it is rarely accompanied by an open-source documentation license. Documentation for open-source software projects often remains proprietary, and resistant to external contributions.

The Linux Documentation Project, as an example, explicitly prohibits open use of the documentation source without written permission:

> Any translation or derivative work of Linux Installation and Getting Started must be approved by the author in writing before distribution. … These restrictions are here to protect us as authors, not to restrict you as learners and educators.
>
> – Linux Documentation Project Copying License [11]

While many open-source projects do have a more relaxed approach to copyright and some use clearly open licenses, in reality few members of open-source software projects participate in the development of documentation and the writing staff is a relatively limited group of people. The most open-source documentation projects can be found in the online annotated and editable manuals, for instance the PHP annotated manual [23], the MySQL commented manual [14], and the Squeek editable manual [35]. These systems allow users to comment on, or in the case of Squeek, directly edit, the documentation. The licensing policy is, however, unclear or closed in these examples, and there is no explicit social structure to aid in assessing contributors' credibility.

Discussion forums and mailing lists provide a high degree of user control, flexibility, and openness to contributions. The members of the community easily participate. User control over content is built in to the submission structure. Release cycles can be very short as answers to questions are posted often within hours. Splits are not uncommon into different strands of continued discussion. Unfortunately, discussion forums and mailing are lists have difficulty supporting the task of building documentation. Extraction of material into documentation is seldom performed, making discussions concerning topics difficult to track. Search engines do exist for such purposes, but require a common terminology across submissions and support only active search (not passive browsing).

To find really good examples of open-source documentation we have to look at more general projects. A well known example from the software world is Slashdot (www.slashdot.org), which has been around since 1997 and where the majority of the work is done by the people who e-mail stories to the site [31]. Slashdot puts a strong focus on documentation development through moderated discussion, but an explicit open-source documentation policy is still lacking and there is little focus on building documentation.

Even more developed open-source documentation projects can be found outside the software world. The Nupedia [16] and Wikipedia [38], globally written encyclopaedias, are examples of projects that develop information using the GNU Free Documentation License and that provide a social structure for writers and editors. In many ways these projects can be viewed as being open-source documentation projects.

In conclusion, many open-source documentation projects today are not really open, even in open source software projects. What is lacking is largely an open-source documentation license policy, explicit social structures and documentation builds. To a certain degree human resources are also lacking: for instance, open source software projects have not really focused their resources on documentation. The strongest existing examples are general in nature and are not concerned with producing documentation for specific software systems or development projects.

7. WOULD IT WORK

In this paper we have discussed open-source development as a production model that results in user-driven, just-in-time content. We have provided a framework for open-source documentation projects that illustrates what aspects of development need to be taken into account. Furthermore, we have examined open writing techniques and the current state of the profession in real open-source documentation projects.

Open-source documentation may well be an attractive method for user-driven, just-in-time production of documentation, in particular seeing as much of the production is performed free of charge. However, that does not mean it will work. The fact that most of the software needed for handling open-source documentation projects already exists for open-source software development is advantageous. However, documentation has its own problems that do not exist in the software realm. For instance, changing the documentation does not change the functionality of the software, and incorrect content is not as easily caught by compilers and test cases. Greater care and more review may be required for open-source documentation compared to open-source software.

It is also important to remember that the completeness of the open-source documentation project may not be the ultimate goal. Documentation should provide answers to user questions and does not need to totally describe the system. Let's put it another way: the absence of description in an open-source documentation project may in itself be a source of knowledge. If users do not request documentation for a particular feature, it may be because the answer is made obvious by the design of the interface, or the feature may simply not be used (assuming that users faithfully report their needs). In the latter case, the hole in the documentation may soon have a matching hole in the software! Using an open-source documentation process provides a way to measure areas of use and kinds of interaction, and may therefore be valuable to the development process. As much as users are involved in the documentation process by providing discussion content, asking questions and answering them, they are also providing requirements for tomorrow. What users question and provide answers for can demonstrate what parts of the software they use.

What will become of the writing staff in an open-source documentation project? The writing staff should be dedicated members of the open-source projects. Given that a large enough user community exists, the writing staff would service the writing

community with their expert knowledge about the system and help developers articulate themselves. Gate-keeping the production of content becomes a vital task. Furthermore, the writing staff should create documentation by extracting content that passes though mailing lists and discussion forums: FAQs, development documentation and technical manuals. Such content extraction would serve both the documentation and the development process. If fewer users contributed, the writing staff would need to increase their original content production.

Success ultimately depends on the open-source documentation project's ability to accumulate enough users that can and will contribute to the process. Open-source software has shown that it is possible to generate even large applications from the efforts of users. Projects such as Nupedia have also shown that this fact translates to open source documentation. However, smaller projects may have difficulties producing enough user contribution. On the other hand, let us not forget that users definitely can provide questions even when they can't provide answers. In this sense, open-source documentation provide much needed relevance and priority assessments to the documentation process.

8. REFERENCES

[1] Apache (open-source web server) http://www.apache.org

[2] Carroll J. M. (Ed.) (1998) *Minimalism Beyond the Nurnberg Funnel* MIT Press 1998, Cambridge, Massachusets.

[3] Debian (open-source Linux) http:// www.debian.org

[4] DiBona C., Ockman S., and Stone M. (Eds.) (1999) Open Sources: Voices from the Open Source Revolution O'Reilly

[5] Feller J., and Fitzgerald B. (2000) *A Framework Analysis of the Open Source Software Development Paradigm* In Proceedings of the 21st International Conference on Information Systems 2000, Brisbane pp. 10-13

[6] Feller J., Fitzgerald B., and van der Hoek, A. (2001) *(W18) 1:st Workshop on Open Source Software Engineering*, position paper for the workshop In Proceedings of the 23rd International Conference on Software Engineering, 2001 pp. 780 –781

[7] GNU Emacs (open-source extensible editor) http://www.gnu.org/software/emacs/

[8] GNU Free Documentation License http://www.gnu.org/copyleft/fdl.html

[9] GNU Software (original free software initiative, origin of open-source) http://www.gnu.org

[10] Jones P. (2001) *Open(sourcing) the Doors: for Contributor-Run Digital Libraries* Communications of the ACM vol. 44. no. 5 pp. 45-46

[11] Linux Documentation Project copying license, viewed August 2001. http://www.linuxdoc.org/LDP-COPYRIGHT.html

[12] Linux.org (central source of Linux information) http://www.linux.org

[13] Mozilla (open-source web browser, development project for Netscape 6, based on the original Netscape source code) http://www.mozilla.org

[14] MySQL annotated manual (online annotated manual) http://www.mysql.com/doc/

[15] Netcraft Web Server Surveys, viewed June 2001 http://www.netcraft.com/survey/

[16] Nupedia (open-source encyclopedia) http://www.nupedia.com/

[17] Open Content License http://www.opencontent.org

[18] Open Source Initiative http://www.opensource.org

[19] Open Source Initiative, definition of open-source http://www.opensource.org/docs/definition.html

[20] Perence B. (1999) *The Open Source Definition* in Open Sources: Voices from the Open Source Revolution, Eds. DiBona C.,

[21] Perkins (1999) *Culture Clash and the Road to World Domination* IEEE Software January/February 1999 pp. 80–84

[22] Perl (open source programming language) http://www.perl.org

[23] PHP annotated manual (online annotated manual) http://www.php.net/manual/en/

[24] Priestley, M. (2001) *DITA XML: A Reuse by Reference Architecture for Technical Documentation* Conference Proceedings, ACM SIGDOC 2001

[25] Priestley, M., and Utt, M. H. (2000) *A unified process for software and documentation development* Conference Proceedings, IEEE/ACM IPCC/SIGDOC 2000

[26] Python (open source programming language) http://www.python.org

[27] Raymond E. S. (1999a) The Cathedral and the Bazaar, *O'Reilly*

[28] Raymond E. S. (1999b) *A Brief History of Hackerdom* in Open Sources: Voices from the Open Source Revolution, Eds. DiBona C., Ockman S., and Stone M., O'Reilly

[29] Raymond E. S. (1999c) *Linux and Open-Source Success* (interview) IEEE Software January/February 1999 pp. 85–89

[30] Sanders J. (1998) *Linux, Open Source, and Software's Future* IEEE Software September/October 1998 pp 88–91

[31] Slashdot (open software e-zine) viewed August 2001 http://slashdot.org/about.shtml

[32] SourceForge (web site for multiple open source projects) http://sourceforge.net/

[33] Stallman R (1999) *The GNU Operating System and the Free Software Movement* in Open Sources: Voices from the Open Source Revolution, Eds. DiBona C., Ockman S., and Stone M., O'Reilly

[34] Stanford Copyright and Fair Use web site http://fairuse.stanford.edu

[35] Squeek manual (online editable manual) http://squeak.cs.uiuc.edu/documentation/index.html

[36] Utt, M.H., and Mathews, R. (1999) *Developing a User Information Architecture for Rational's ClearCase Product Family Documentation Set*. Conference Proceedings, ACM SIGDOC 1999, pages 86-92.

[37] Vixie P. (1999) *Software Engineering* in Open Sources: Voices from the Open Source Revolution, Eds. DiBona C., Ockman S., and Stone M., O'Reilly

[38] Wikipedia (open-source encyclopedia) http://www.wikipedia.com

[39] Willson (1999) *Is the Open-source Community setting a Bad Example?* IEEE Software January/February 1999 pp. 23–25

9. ABOUT THE AUTHORS

Erik Berglund is a Ph.D. candidate at the Department of Computer and Information Science at Linköping University, Sweden. He works with reference documentation and communication in programming and has previously published at SIGDOC. Current topics include global sharing and community communication.

Michael Priestley is an information developer at the IBM Toronto Software Development Laboratory. He has written numerous papers on subjects such as hypertext navigation, singlesourcing, and XML publishing systems. He is currently working on XML and XSL prototypes for help and documentation management, and is the Specialization Architect for the Darwin Information Typing Architecture.

This paper represents the views of the authors and not necessarily those of their employers.