Self-organization of teams for free/libre open source software development

Abstract

This paper provides empirical evidence about how free/libre open source software development teams self-organize their work. Following a case study methodology, we examined developer interaction data from three active and successful FLOSS projects using qualitative research methods, specifically inductive content analysis, to identify the task-assignment mechanisms used by the participants. We found that 'self-assignment' was the most common mechanism across three FLOSS projects. This mechanism is consistent with expectations for distributed and largely volunteer teams. We conclude by discussing whether these emergent practices can be usefully transferred to mainstream practice and indicating directions for future research.

(96 words)

Keywords: Free/libre open source software development, task assignment, self-organizing teams, distributed teams, qualitative research methods

Running head: Self-organization of teams in FLOSS development

Self-organization of teams for free/libre open source software development

1. Introduction

Researchers have increasingly realized that large-scale software engineering is a social activity involving numerous developers and other professionals working closely together in a tightly coordinated process. In order to understand this social activity, researchers have expanded their focus to include not just the code, but also the processes and social practices that create it. To better understand the social side of software development, we apply empirical methods that aim at understanding, rather than just measuring, development practices.

We are particularly interested in the problems of distributed development, because distributed groups are increasingly used in software development but face particular challenges to their effective work. To develop insights into the issues faced in distributed development, we examine an extreme version, namely the development of free/libre open source software[1] (FLOSS). FLOSS is a broad term used to embrace software developed and released under an "open source" license allowing inspection, modification and redistribution of the software's source without charge ("free as in beer"). Much though not all of this software is also "free software", meaning that derivative works must be made available under the same unrestrictive license terms ("free as in speech", thus "libre"). Characterized by a globally distributed developer force and a rapid and reliable software development process, effective FLOSS development teams somehow profit from the advantages and overcome the disadvantages of

1 The free software movement and the open source movement are distinct and have different philosophies but mostly common practices. The licenses they use allow users to obtain and distribute the software's original source code, to redistribute the software, and to publish modified versions as source code and in executable form. While the open source movement views these freedoms pragmatically (as a development methodology), the Free Software movement regards them as human rights, a meaning captured by the French/Spanish word 'libre' and by the saying "think of free speech, not free beer". (See http://www.gnu.org/philosophy/ and http://opensource.org for more details.) This paper focuses on the development practices of these teams, which are largely shared across both movements. However, in recognition of differences between these two communities, we use the acronym FLOSS, standing for Free/Libre and Open Source Software, rather than the more common OSS.

distributed work [2], making their practices potentially of great interest to mainstream software development.

This paper makes the following contributions. First, it provides empirical evidence about management practices of FLOSS teams. Specifically, we identify how FLOSS team self-organize their work, how these practices differ from those of conventional software development and thus suggest what might be learned from FLOSS and applied in other settings. Second, the paper provides an example of the application of qualitative research methods, specifically inductive content analysis, to software engineering research.

The paper continues in five sections. We first briefly review literature on distributed software development and on FLOSS in particular to identify what is known or believed about FLOSS development practices, in order to motivate our research question. We then discuss our qualitative case-based research method, followed by the findings from our analysis. After discussing the implications of our findings, we conclude by considering implications for software engineering in general and possibilities for future research.

2. The challenges of distributed software development

Though distributed work has a long history [e.g., 36], advances in information and communication technologies have been crucial enablers for recent developments of this organizational form [1]. Distributed teams seem particularly attractive for software development because the software source code and other artifacts can be shared via the same systems used to support team interactions [35, 41]. While distributed teams have many potential benefits, distributed workers face many real challenges. Watson-Manheim, Chudoba, & Crowston [48] argue that distributed work is characterized by numerous discontinuities: a lack of coherence in some aspects of the work setting (e.g., organizational membership, business function, task, language or culture) that hinders members in making sense of the task and of communications

from others [46], or that produces unintended information filtering [16] or misunderstandings [3]. These interpretative difficulties in turn make it hard for team members to develop shared mental models of the developing project [15, 18]. A lack of common knowledge about the status, authority and competencies of team participants can be an obstacle to the development of team norms [4] and conventions [31].

The presence of discontinuities seems likely to be particularly problematic for software developers [46]. Numerous studies of the social aspects of software development teams [14, 25, 40, 46, 47] conclude that large system development requires knowledge from many domains, which is thinly spread among different developers [14]. As a result, large projects require a high degree of knowledge integration and the coordinated efforts of multiple developers [8]. More effort is required for interaction when participants are distant and unfamiliar with each others work [37, 43]. The additional effort required for distributed work often translates into delays in software release compared to traditional face-to-face teams [23, 34]. The problems facing distributed software development teams are reflected in Conway's law, which states that the structure of a product mirrors the structure of the organization that creates it. Accordingly, splitting software development across a distributed team will make it hard to achieve an integrated product [22].

2.1. FLOSS development as distributed software engineering

While the literature reviewed above highlights the difficulties involved in distributed software development, the case of FLOSS development provides a striking counter-example. There are thousands of successful FLOSS projects, spanning a wide range of applications, most developed by distributed teams. Due to their size, success and influence, the Linux operating system and the Apache Web Server and related projects are the most well known, but hundreds of others are in widespread use, including projects on Internet infrastructure (e.g., sendmail,

bind), user applications (e.g., Mozilla, OpenOffice) and programming languages (e.g., Perl, Python, gcc). Many are popular (indeed, some dominate their market segment) and the code has been found to be generally of good quality [44]. The success of these projects in managing distributed development raises the question of what can be learned from this setting and applied to software development and distributed work more generally.

As well, FLOSS development is an important phenomena deserving of study for itself. FLOSS is an increasingly important commercial phenomenon involving all kinds of software development firms, large, small and startup. Millions of users depend on systems such as Linux and the Internet (heavily dependent on FLOSS tools), but as Scacchi [42] notes, "little is known about how people in these communities coordinate software development across different settings, or about what software processes, work practices, and organizational contexts are necessary to their success". As well, understanding FLOSS development teams is important as they are potentially training grounds for future software developers.

The nascent research literature on FLOSS has addressed a variety of questions. First, researchers have examined the implications of FLOSS from economic and policy perspectives. For example, some authors have examined the implications of free software for commercial software companies or the implications of intellectual property laws for FLOSS [17, 27, 28]. Second, various explanations have been proposed for the decision by individuals to contribute to projects without pay [5, 19, 20, 24, 32]. These authors have mentioned factors such as personal interest, ideological commitment, development of skills [29] or enhancement of reputation [32]. Finally, a few authors have investigated the processes of FLOSS development [e.g., 38, 45], which is the focus of our study.

Raymond's [38] bazaar metaphor is perhaps the most well known model of the FLOSS process. As with merchants in a bazaar, FLOSS developers are said to autonomously decide how

and when to contribute to project development. By contrast, traditional software development is likened to building a cathedral, progressing slowly under the control of a master architect. While popular, the bazaar metaphor has been broadly criticized. According to its detractors, the bazaar metaphor disregards important aspects of FLOSS development processes, such as the importance of project leader control, the existence of de-facto hierarchies and emergent leadership, the danger of information overload and burnout, and the possibility of conflicts that cause a loss of interest in a project or forking [6, 7].

In this paper, we examine the practices of FLOSS projects in more detail to provide a richer picture of how these teams accomplish software development. The archetypical community-based FLOSS development process[2] differs from proprietary development in several respects that affect or depend on the approach used for managing the project. A primary difference is that the community-based development process is not owned by a single organization. Developers contribute from around the world, meet face-to-face infrequently if at all, and coordinate their activity primarily by means of computer-mediated communications (CMC) [38, 49] and other software development tools (e.g., source code control systems).

What is perhaps most surprising about the FLOSS process is that it appears to eschew traditional project coordination mechanisms such as formal planning, system-level design, schedules, and defined development processes [22]. Many teams are largely self-organizing, without formally appointed leaders or indications of rank or role, raising the question of how the work of these teams is managed. In addition, non-member involvement plays an important role in the success of the teams. Generally, a small core group oversee the overall design and contribute the bulk of the code, but other developers play an important role by contributing bug fixes, new features or documentation, by providing support for new users and filling other roles

in the teams. Core group membership can bestow some rights, including deciding what features should be integrated in the release of the software, when and how to empower other code maintainers, or to "pass the baton" to the next volunteer [39]. However, in comparison to traditional organizations, more people can share power and be involved in group activities. In most projects, anyone with enough interest and skill can access the code, contribute patches, make suggestions to group, and attend important decision processes. Users who are non-members or peripheral members become a crucial resource of potential recruitment [21]. How to handle the relationship between non-members' requirements and the project goal is thus a significant challenge.

These features make FLOSS teams extreme examples of self-organizing distributed teams, but they are not inconsistent with what many organizations are facing in recruiting and motivating professionals and in developing distributed teams. As Peter Drucker put it, "increasingly employees are going to be volunteers, because a knowledge worker has mobility and can go pretty much every place, and knows it… Businesses will have to learn to treat knowledge workers as volunteers" [9]. These characteristics of self-organization and volunteerism make FLOSS teams particularly interesting sites for studying the social side of software engineering practices.

2.2. Research question

The research reviewed above suggests that distributed teams should face significant problems in developing coherent software, but that self-organizing FLOSS teams have had some success in doing so. As well, the literature on FLOSS development in particular suggests that many of these teams adopt a very different approach to organizing the contributions of team

[2] In focusing on community-based development, we exclude projects such as MySQL that are developed by a single organization following a conventional software engineering approach and only released under a FLOSS license.

members. Therefore, in order to shed more light on these practices and how they address the challenges of distributed development, we analyze the process by which specific developers get assigned to work on parts of the project development, such as particular patches or bug fixes. Our empirical work, described next, addresses one side of the comparison.

For the other side, we draw on published descriptions of task assignment in proprietary software development. For example, in the bug fixing process described by [10], developers are assigned to work on particular modules of code, meaning that bugs in those modules must be routed to that engineer to work on. In order to assign tasks then, customers give problem reports to the service centre, which in turn assigns the problems to product engineers, who then assign them to software engineers. In addition, software engineers may assign reports or subtasks to each other. Specialization allows developers to develop expertise in a few modules, which is particularly important when the engineers are also developing new versions of the system, and allows a single person to manage all changes to particular modules, thus minimizing the cost of integrating changes. However, the cost of such a system is the need for an elaborate process for assigning tasks to the appropriate developer. In our empirical work, we examine the nature of the corresponding process for FLOSS development.

3. Methods: Inductive coding of FLOSS developer email interactions

In this section we describe the approach we adopted to analyze task assignment in FLOSS software development processes. Because the task assignment mechanisms used by these teams had not yet been described, we adopted an inductive multiple case study approach to the research. Rather than starting with hypotheses to be supported or reject, this approach involves developing new theory to address our research question by carefully analyzing real FLOSS developers' interactions for evidence of the task assignment mechanisms in use. An inductive approach was indicated by our desire to extend theory for this new phenomenon. The rationale

for the use of a case study approach was that case studies provide rich empirical data from a real setting, necessary for theory generation. In particular, Yin [51] notes that case studies are particularly appropriate for answering "how" or "why" questions about current events in situations where the researcher has no control over the circumstances of the study.

3.1. Sample selection

Because the process of manually reading, rereading, coding and recoding messages is extremely labor-intensive, we had to focus our attention on a small number of projects. (Overcoming this limitation is a subject of future research, as discussed below.) A theoretical sampling strategy was employed in this study, meaning that we selected case sites based on their expected contributions to theory development rather than for representativeness (a criterion that would have been appropriate for theory testing). We adopted several criteria for choosing projects. First, the data from these projects that we needed for analysis had to be publicly available (ruling out projects that limit access to their email lists or trackers). Second, we chose the projects that had more than 7 members, because small projects are less likely to assign tasks in an observable way or to have significant task assignment problems. Third and most important, we wanted to study that projects that seem to be relatively successful at managing the contributions of multiple developers (the core developers plus many more peripheral contributors), thus providing relevant data for insight into task assignment mechanisms for FLOSS development. We assessed success according to the criteria suggested by Crowston et al. [12], looking for projects that have attracted numerous developers beyond the initial project founders, are continuing to release software, have numerous downloads and have an active user community that provides feedback.

Based on these criteria, 3 FLOSS projects were selected for analysis, namely Gaim, eGroupWare and Compiere ERP.

- Gaim is an instant messenger application that supports multiple platforms and protocols (http://sourceforge.net/projects/Gaim/).
- eGroupWare is a multi-user, web-based groupware suite with modules such as email, address book, calendar, content management, forum, wiki and so on (http://sourceforge.net/projects/eGroupWare/).
- Compiere is an ERP+CRM solution for Small-Medium Enterprises covering areas such as order and customer/supplier management, supply chain and accounting (http://sourceforge.net/projects/compiere/ and http://www.compiere.org/).

The development status of the three projects is shown in Table 1. Note that all three projects are hosted on the SourceForge system (http://sourceforge.net/), which controls for differences attributable to accessibility or choice of development tools.

Insert Table 1 about here

Despite similarities in status, these projects differ in ways that allows for some interesting comparisons. Gaim is an end-user desktop application written in C, while eGroupWare in a web-based server application written in PHP. As a result, we expect Gaim mostly to be used as is, but expect eGroupWare to have a lower barrier to entry for developers and to be more often customized, potentially increasing problems in integrating these contributions. Compiere was originally a commercially developed product that later moved to Open Source development. Its history allows us to examine issues that arise as a new set of developers began working with an established code base and developer community. As well, one of the authors had extensive experience with proprietary ERP systems, providing a basis for comparison.

3.2. Data

The primary data used for our study were interactions on the main developer communication forum, either a developer mailing list or web-based discussion forum. We chose these interactions because they contain the communications between developers used to coordinate project development. We also used the list of developers that appears on the project's home page. To increase the comparability of our analyses of each project, we examined messages from a similar period in the project development lifecycle. We expected that management-related activities would occur more frequently around (and especially before) a major release, so we analyzed the period leading up to and immediately after the first open source release for each project. Specifically:

- For Gaim, we selected messages posted to the "Gaim-devel" mailing list during August and September 2004 for analysis (Gaim 1.0.0 was released on 16 September 2004). The total number of messages was 710, posted by 85 individuals (11 were identified as developers according to the current developer list for Gaim, and 1 was identified as a former developer).
- For eGroupWare, we selected messages posted to the "eGroupWare-development" mailing list during October and November 2004 for analysis (version 1.0.00.006 of eGroupWare was released on 18 November 2004), which resulted in 665 messages total posted by 151 individuals (20 were identified as developers according to the current developer list).
- For Compiere, we selected messages posted to the Development Chat Forum from the January 1, 2001 to November 20, 2002, which covers the period up to and following the 5 Sept 2002 release of the Compiere Version 2.4.3a, the first version released after the move to SourceForge. Perhaps because this was a project transitioning from in-house

development, the initial traffic on the developers' forum was sparse and we had to draw from a longer time period than with the other projects to gather the 315 messages we examined. The Compiere project had more defined roles: there were postings from 57 users, from 2 project managers, from 3 advisor/mentor /consultants, from 6 developers and from 3 translators.

A consequence of our choice of these messages is that the dates of the messages and date of the list of developers are not the same. Because the list of developers can change over time, it is difficult to tell in a retrospective analysis exactly what an individual's status was at the time a message was posted. We do not believe that changes in the developer lists substantially affect our conclusions, but we plan to develop time-stamped lists of developers for future research.

As mentioned above, all three projects are hosted on SourceForge, making data about them easily accessible for analysis. Nevertheless, analysis of these data poses some ethical concerns that we had to address in gaining human subjects approval for our study. On the one hand, the interactions recorded are all public and developers have no expectations of privacy for their statements (indeed, the expectation is the opposite, that their comments will be widely broadcast). Consent is generally not required for studies of public behaviour. On the other hand, the data were not made available for research purposes but rather to support the work of the teams. We have therefore followed the common practice of rendering all data anonymous by using pseudonyms in publications.

3.3. Analysis

For this project, we inductively content-analyzed developer email interactions to identify the task assignment mechanisms used in the process. We coded each instance of task assignment identified on three dimensions: who assigned the task, to whom, and how. For the first two dimensions, we identified two main types of actors in FLOSS projects from our literature review:

developers, which we defined by the list of developer named on the project's Sourceforge page, and users, all those whose names are not listed. Tasks can thus be assigned in four directions: from developer to developer, from user to user, from developer to user and from user to developer.

To address how tasks are assigned, three coders independently went though the data and identified all the behaviors related to task assignment. We started by developing a coding scheme based on prior work on coordination modelling [13], which provided a template of expected activities needed for task assignment. Specifically, we looked for evidence of actions taken to identify a task that need to be performed, identification of which individuals could perform the task, selection of a particular individual and some kind of assignment of the task to the individual. For the first, we looked for statement of intention to undertake or perform a task, rather than a status or process report. For example, if someone said "I'd like to do it", we considered this statement part of a task assignment. But if he or she said "I did it" or "I have already committed the code successfully", then we did not consider this statement as a task assignment. In our qualitative analysis, we observed many messages beginning with "`I have already done it`", or "`I spent three days working on this bug and solved it finally`", even though no previous postings could be found to discuss how or to whom the task was assigned. We did not code such statements because these status/process reports show who did a task, but do not indicate anything about the prior steps in task assignment, such as who identified or assigned the task or how it was assigned.

The coding system evolved through discussion of the applicability of codes to particular examples, both in group meetings and as three coders worked on the same set of messages over the course of several months. The final coding scheme is shown in Table 2. All messages were double-coded to allow computation of inter-rater reliability. The level of inter-rater agreement

for coding the Compiere, eGroupWare and Gaim projects were 0.893, 0.887 and 0.810 respectively, all above the usual rule-of-thumb acceptable level of 0.80.

Insert Table 2 about here

4. Results

In this section, we discuss the results of our analysis. We discuss first the use of different kinds of task assignment, then consider who does the assignment and to whom. For the first, the frequencies of the five task assignment mechanisms in the three projects are shown in Table 3.

Insert Table 3 about here

Our first finding confirms a striking feature of FLOSS development practices, which is how often developers introduce a task and simultaneously offer to work on it, in effect assigning the task to themselves. As Table 3 shows, for all three projects, self-assignment is the most frequent form of assignment. The following is an example of a poster on the eGroupWare mailing list self-assigning a task:

> Thanx for the wonderfull help ... Maybe an idea to make a 'hello world' package for on the website ... this kind of standard app .. would be verry Helpful for starting eGW developers (like me ;-)) If you'd like, I'll make the package.

The volunteering may be coupled with an inquiry about the usability of product. This example came from the Compiere project for example:

> I want to extend and to create new reports of Compiere, I know that you have thought to replace to Style Report by API Java 1.4. Which is your plan to implement API Pringing?

In a few cases, the offer to help is not connected to a particular task, but rather to claim responsibility for a general class of problems or just to announce availability, as in these two quotations:

> Hi, developers, If you find any PHP5 related problems, please open a bug report and assign it to me.
>
> Well I just started 14 days of vecation in front of the telly and computer. So I guess I will do some coding these days. =)

Due to the nature of voluntary participation, conditional volunteering behaviours are frequently observed in all three projects. Unlike in most proprietary software development, volunteers in FLOSS projects take on the work on their own time. For example, it is common to see "I would like to work on it, if I get time". And the volunteers are not required to be an expert, or even familiar with the task he wants to take on. Volunteering combined with asking for help is frequently seen in our projects, as shown in these two examples:

> I'd willing to do this, but really need some assistance upfront before I could make a contribution. I was wondering if someone here might be willing to help me.
>
> Can someone give me instructions to translate, so I can work on that?

As well as volunteering, developers often propose tasks and ask for volunteers, explicitly or implicitly. For eGroupWare and Compiere, asking a certain person was the second most frequent mode of task assignment, followed by asking an unspecified person. For Gaim, ask an unspecified person is the second most frequent mode of task assignment, followed by asking a certain person. For example:

> Can someone please do a brief test, replacing config.php with newconfig.php? If it works for a few people without causing problems, it will help us in the long run.

Sometimes asking for volunteers is coupled with volunteering, as in these two quotations:

> Our own project repository would require maintenance, bandwidth, and drive space. I've volunteered to do everything to get us started. Volunteers to help maintain would be appreciated.
>
> I'm currently working through a port of my Bugzilla data into TTS, so feel free to ask me for any more tips, suggestions, etc. I'm happy to help, as I'm going through the process myself!

Strikingly, we observed almost no cases of someone asking a non-team member to work on some part of the project.

Our third finding is that sometimes team members are suggested to consult with others before they do a certain task, especially for eGroupWare. For example:

> I only ask you to consult with the maintainer of the concerned app before you commit something. If you need to change something radical in the API please talk it through Lars or me before (eg. Mail it as proposal to the developers list).
>
> Sorry for that. Have you talked to Bill about the patches? I can't image he's not willing to accept them.

Sometimes people don't directly ask others to do a task, but discuss who can do among several candidates. For example:

> Miguel our translation coordinator or I will get the translation via your link and will commit them / include them in the distribution. Ronald
>
> Amir: are you going to commit the translations and add Thai to setup/lang/languages.php to enable Thai or should I take care of that?

Second, since self-assignment depends on the ability of an individual to contribute to the group, we investigated differences between developers and users in the type of assignment used, as shown in Table 4. As noted above, the status of user or developer was assigned by comparing the poster of the message to the project's published list of developers.

Insert Table 4 about here

The table shows that for all three teams, users rarely assign work to other users, but still often self-assign tasks. Especially for Gaim the percentage of users doing self-assignment is very high and some users offer to contribute several times. For example, one user in eGroupWare committed himself to helping implement features:

> I am willing to help implement this feature, and have sometime free-time:-) to give away, so I started looking through the code.

In contrast to the separation of roles in the proprietary process, many users who post to the developer email list prefer to solve the problem by themselves when they identify a task, instead of reporting it directly and just waiting for the responses. In other words, the formal division of users and developers does not necessarily constrain their behaviors.

Though the previous tables show many similarities among three cases, some interesting differences in task assignment mechanisms were found through in-depth qualitative analysis. Most of them are closely related to the nature of products and characteristics of projects. For example, unlike Gaim and eGroupWare, Compiere was originally a commercially initiated project. As well, its target users are Small to Medium Enterprises. Compared to the other projects, self-assignment looks more like the result of group decision-making in Compiere. Both developers and users tend to use "We" instead of "I" most of the time, such as:

> We will give it try and hope that you will contribute or help us during the coding process.

In addition, the percentage of conditional task assignment is much higher in Compiere than the other two, which we attribute to the complexity and size of code, and the fact at the time of our analysis, new volunteers were just beginning to work with what had been a proprietary code base. More than half of volunteering work asks for help before committing to contribute. On the other hand, these kinds of messages were rarely founded in self-assignment activities in eGroupWare, because it has a lower barrier to newcomers as we expected. Furthermore, the coupling of asking for volunteers and volunteering is also more frequently used in Compiere than Gaim and eGroupWare. Most of the requirements from business customers are relatively complex and hard for one individual to implement. As a result, developers tend to post tasks publicly, commit to do it and try to attract more volunteers.

5. Discussion

The most striking difference we noted between FLOSS and proprietary software development is that community-based FLOSS development does in fact seem to rely less on explicit assignments of work, as has been suggested in the literature on FLOSS development. We did not observe evidence of a "hierarchy" in assigning tasks in FLOSS. By lack of hierarchy, we mean that individuals do not command or direct others to work on a task as might a project manager or employer. Instead, they use phrases such as, "`would you please`", "`if you have time, can you`…" or even discuss how to assign a task instead of simply assigning it directly. Even assignment to a specific person is qualified: "`If you're interested you can …`" or "`feel free to figure out why and fix it if you like.` " Similar differences have been noted in the Linux process, which also relies on developers assigning themselves to tasks rather than being explicitly assigned [11].

Overall, the FLOSS task assignment process seems similar to the market approach to task assignment suggested by Crowston [10]. In a market form of task assignment, a description of each task is sent to all available agents. Each evaluates the task and, if interested in working on it, submits a bid, saying how long it would take, how much it would cost or even what they would charge to do it. The task is then assigned to the best bidder, thus using information supplied by the agents themselves as the basis for picking who should work on the task. However, the FLOSS process differs in that there is often no separation of the identification of task and its assignment, nor is there always an explicit, or authoritative, assignment of the task. The use of a market-like task assignment mechanism in the FLOSS process is consistent with predictions of the effect of the more extensive use of information and communication technologies (ICT). ICT makes it easier to gather information about available tasks and resources and to distribute the decision about which resources to use for a particular task. At a macro level,

Malone, Yates and Benjamin [30] suggest that decreased coordination costs favor more extensive use of markets, which usually have lower costs but require more coordination activities, over vertical integration, which makes the opposite trade-off. In contrast, for the proprietary process, the choice is often made by a project manager based on the specializations of the individual developers [10].

Second, we observed broader participation in the work on tasks. Due to the openness of the FLOSS teams, active users can take on development tasks rather than the process being restricted to the official development team. Indeed, for all three teams, the most common form of task assignment was self-assignment, that is, volunteering to work on a particular task among both developers and users. These active users do not wait to be given something to do, but rather step forward to work on tasks that catch their interest. One likely explanation for this difference is that FLOSS development teams are substantially composed of volunteers. As a result, task assignment in FLOSS needs to be based primarily on personal interest.

As well, self-assignment appears to play an important role in bolstering the legitimacy of the action suggested by the poster, making it more likely that the group will consent to the poster's preferences. Legitimacy is the right of a participant to be heard and respected in a forum and a source of influence on that forum's decisions. Formal groups with pre-defined memberships assess legitimacy up-front, before admitting members. In proprietary software development the right to make suggestions for features is often reserved for the specification writers or customer facing "marketing engineers". On the other hand, for informal groups seeking new contributors it is not possible, nor desirable, to make such pre-qualifications. The prevalence of self-assignment supports the anecdotal suggestion that in FLOSS projects in general, legitimacy is linked to action, the value that "code speaks louder than words". After all, if someone is willing to use their own time to implement a feature or fix a bug (or even better, if

they have actually already done so) it is more difficult for other members of the team to deny them that opportunity.

Of course, reliance on this kind of assignment does have several drawbacks. First, anyone can choose to contribute to the project, even if they are not good at it or have no relevant experience. As a result, the quality of team member output often needs further investigation by developers and other users. In some projects, developers chose to forego these contributions rather than to spend the time evaluating the output. Second, because multiple developers may be working on the same parts of the project, projects must develop code management practices that allow multiple changes to be integrated. Many projects rely on code management systems such as CVS, and the design of these tools has become a pressing topic in FLOSS discussions (see the recent discussion of source code control tools for Linux summarized at http://en.wikipedia.org/wiki/BitKeeper). We did observe some interesting behaviors to avoid duplication before volunteering to do a task, such as the following:

> can I can dibs on this? I don't want to have any of your work duplicated, so I want to make sure that I don't infringe on what someone is already working on.

The use of self-assignment may be an emergent phenomenon in the teams, consistent with a characterization of the teams as self-organizing. For example, the quotation above begins with the poster's disclaimer, "I don't know how assignments work, but can I can [sic] dibs on this?" In the absence of guidelines or rules, self-assignment may emerge to fill the need for a coordination mechanism to manage this part of the collaboration.

Finally, proprietary task assignment and resource planning relies on the participants being reliable; the employment relationship, with its clear penalties for non-performance gives managers a reliable expectation that the work assigned will be carried out. By contrast, it has been suggested that FLOSS participants are relatively unreliable [33]. The data in this study

provide some support for this idea, though we did not directly measure the motivations or volunteer status of the participants so this cannot be fully confirmed. Much of the self-assignment is qualified with phrases like "`if I have time`", "`if I get some free time`" or which cite 'real world' time constraints, "`I will submit this in a couple of day (have to finish some professional work first ... :/)`". This uncertainty can create problems in the usability of output and timeliness of schedule. People finish their tasks according to time and interest, which may affect the effectiveness of the project. And because of uncertainty of code ownership, and the distributed nature of the participants, it is hard to track the status of developers' work, especially when it is self-assigned. These trade-offs make sense in a volunteer activity where the service of individuals is on their own terms. However, these characteristics may also be increasingly common in distributed software development, characterized as it is by numerous discontinuities. Discontinuities of place and time mean that the actions of remote colleagues are not easily visible, potentially making their contribution seem less reliable. As well, different team members may work for different organizations (i.e., across another discontinuity), again making them increasingly similar to FLOSS teams. As a result, the approaches used by FLOSS teams may be appropriate even in non-FLOSS settings.

6. Conclusion

In this paper, we have shown how qualitative inductive analysis provide insights about the differences between FLOSS and proprietary development and thus suggests practices that might be useful for proprietary development. Our results suggest several avenues for future research. First, our results are based on just three projects. Studies of other projects are needed to confirm the pattern of project management reported here and to identify factors affecting the choice of approaches. For example, we expect that task assignment in company-sponsored projects works differently than in community-based projects. Further studies could examine the

role of project culture, leadership or power. Our study has only scratched the surface in this aspect. As well, we have observed that synchronous chat media, such as IRC (Internet Relay Chat), can be another important channel for project management, including task assignment. Due to the unavailability of archives, we were not able to use IRC as a data source in this study, but this channel should be considered in future work. To increase the scope of data collection, we are currently exploring the use of natural language processing techniques to automate aspects of the data analysis process.

Future studies could also examine the link between project management and other group processes. For example, task assignment appears to play a role in the development of shared mental models for the project teams, a key problem confronting software developers as noted above. 'Assignment to unspecified person' provides a discursive and informal but continuous source of to-do items and the desired future of the project, e.g., "`I'd really appreciate it if maybe someone could write the list suggesting the use of 'sound themes.'`" or "`I think it'd be cool if someone modified the image`". Yamauchi et al. [50] identified the practice of maintaining to-dos at various levels of specificity as a feature of FLOSS development. Similarly, 'assignment to a specific person' also communicates the assigner's understanding of the other participants' skills and areas of knowledge. "`Sarah, can you please clarify for me a few things about this design?`" or "`BTW, have you talked with Alvin at all? I think he has something like your global status dropdown somewhat implemented too`", not only creates a task for Sarah or the person asked the question, but announces to the rest of the community that Sarah probably knows something about that design and that Alvin has experience with status dropdowns. In this way practices that build shared mental models are embedded in coordination mechanisms and as such do not require explicit additional work for participants, minimizing the effort required to collaborate.

For managers of traditional software groups, perhaps the most interesting result is use of volunteering as an assignment mechanism. On their face, the concerns noted above about the problems with voluntary assignment suggest that the task assignment practices of FLOSS developers would be hard or undesirable to transfer to mainstream software development. However these coordination mechanisms may not be limited to volunteer projects only. Knoor-Cetina identified similar 'gentle management' in the High Energy Physics (HEP) experiments at CERN. She writes of "a marked self-organization of the experiments observed, a form of voluntarism..." and relates a coordinator using a "gentle" approach in which he "hopefully" finds volunteers ("somebody willing", " because they have a particular interest, something which happens "whenever new tasks require attention." [26 p. 179]. She continues, "it is not 'morale' per se that is [the origin of self-organization or volunteerism], but the discourse that expresses necessities and interdependencies and allows for groups 'freely' (with a little nudging on the part of conveners and spokespersons) to respond to demands". Self-assignment and "gentle" assignment to others, then, is also found within mission-critical work involving the spending of substantial financial and scientific resources. Another example of the use of these techniques within a non-volunteer environment is the practice reported at Google where engineers are allowed one day a week to work on whatever they want to work on (see for example http://www.oreillynet.com/pub/a/network /2005/03 /16/etech_2.html). The innovations in the Google Labs (labs.google.com) are said to be the result of these self-assigned activities. There is scope for the emergent practices identified in this study to be imaginatively applied in other contexts.

8. References

[1] M. K. Ahuja, K. Carley, and D. F. Galletta, "Individual performance in distributed design groups: An empirical study," presented at SIGCPR Conference, San Francisco, 1997.

[2] K. Alho and R. Sulonen, "Supporting virtual software projects on the Web," presented at Workshop on Coordinating Distributed Software Development Projects, 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98), 1998.

[3] D. J. Armstrong and P. Cole, "Managing distance and differences in geographically distributed work groups," in *Distributed Work*, P. Hinds and S. Kiesler, Eds. Cambridge, MA: MIT Press, 2002, pp. 167–186.

[4] D. Bandow, "Geographically distributed work groups and IT: A case study of working relationships and IS professionals," in *Proceedings of the SIGCPR Conference*, 1997, pp. 87–92.

[5] J. Bessen, "Open Source Software: Free Provision of Complex Public Goods," Research on Innovation July 2002.

[6] N. Bezroukov, "A second look at the Cathedral and the Bazaar," *First Monday*, vol. 4, 1999.

[7] N. Bezroukov, "Open source software development as a special type of academic research (critique of vulgar raymondism)," *First Monday*, vol. 4, 1999.

[8] F. P. Brooks, Jr., *The Mythical Man-month: Essays on Software Engineering*. Reading, MA: Addison-Wesley, 1975.

[9] J. Colllins and P. Drucker, "A Conversation between Jim Collins and Peter Drucker," in *Drucker Foundation News*, vol. 7, 1999, pp. 4–5.

[10] K. Crowston, "A coordination theory approach to organizational process design," *Organization Science*, vol. 8, pp. 157–175, 1997.

[11] K. Crowston, "The bug fixing process in proprietary and free/libre open source software: A coordination theory analysis," in *Business Process Transformation*, V. Grover and M. L. Markus, Eds. Armonk, NY: M. E. Sharpe, In press.

[12] K. Crowston, H. Annabi, and J. Howison, "Defining Open Source Software project success," presented at Proceedings of the 24th International Conference on Information Systems (ICIS 2003), Seattle, WA, 2003.

[13] K. Crowston and C. S. Osborn, "A coordination theory approach to process description and redesign," in *Organizing Business Knowledge: The MIT Process Handbook*, T. W. Malone, K. Crowston, and G. Herman, Eds. Cambridge, MA: MIT Press, 2003.

[14] B. Curtis, H. Krasner, and N. Iscoe, "A field study of the software design process for large systems," *Communications of the ACM*, vol. 31, pp. 1268–1287, 1988.

[15] B. Curtis, D. Walz, and J. J. Elam, "Studying the process of software design teams," in *Proceedings of the 5th International Software Process Workshop On Experience With Software Process Models*. Kennebunkport, Maine, United States, 1990, pp. 52–53.

[16] P. S. de Souza, "Asynchronous Organizations for Multi-Algorithm Problems," in *Department of Electrical and Computer Engineering*, : Carnegie-Mellon University, 1993.

[17] C. Di Bona, S. Ockman, and M. Stone, "Open Sources: Voices from the Open Source Revolution." Sebastopol, CA: O'Reilly & Associates, 1999.

[18] J. A. Espinosa, R. E. Kraut, J. F. Lerch, S. A. Slaughter, J. D. Herbsleb, and A. Mockus, "Shared mental models and coordination in large-scale, distributed software development," presented at Twenty-Second International Conference on Information Systems, New Orleans, LA, 2001.

[19] E. Franck and C. Jungwirth, "Reconciling investors and donators: The governance structure of open source," Lehrstuhl für Unternehmensführung und -politik, Universität Zürich, Working Paper No. 8, June 2002.

[20] I.-H. Hann, J. Roberts, S. Slaughter, and R. Fielding, "Economic incentives for participating in open source software projects," in *Proceedings of the Twenty-Third International Conference on Information Systems*, 2002, pp. 365–372.

[21] R. Heckman, Q. Li, and X. Xiao, "How voluntary online learning communities emerge in blended course," presented at Hawaii International Conference on System System (HICSS-39), Kauai, Hawaii, 2006.

[22] J. D. Herbsleb and R. E. Grinter, "Splitting the organization and integrating the code: Conway's law revisited," presented at Proceedings of the International Conference on Software Engineering (ICSE '99), Los Angeles, CA, 1999.

[23] J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter, "An empirical study of global software development: Distance and speed," presented at Proceedings of the International Conference on Software Engineering (ICSE 2001), Toronto, Canada, 2001.

[24] G. Hertel, S. Niedner, and S. Herrmann, "Motivation of Software Developers in Open Source Projects: An Internet-based Survey of Contributors to the Linux Kernel," University of Kiel, Kiel, Germany n.d.

[25] W. S. Humphrey, *Introduction to Team Software Process*: Addison-Wesley, 2000.

[26] K. Knorr-Cetina, *Epistemic Communities*. Cambridge, MA: Harvard Education Press, 1999.

[27] B. Kogut and A. Metiu, "Open-source software development and distributed innovation," *Oxford Review of Economic Policy*, vol. 17, pp. 248–264, 2001.

[28] J. Lerner and J. Tirole, "The open source movement: Key research questions," *European Economic Review*, vol. 45, pp. 819–826, 2001.

[29] J. Ljungberg, "Open Source Movements as a Model for Organizing," *European Journal of Information Systems*, vol. 9, 2000.

[30] T. W. Malone, J. Yates, and R. I. Benjamin, "Electronic markets and electronic hierarchies," *Communications of the ACM*, vol. 30, pp. 484–497, 1987.

[31] G. Mark, "Conventions for coordinating electronic distributed work: A longitudinal study of groupware use," in *Distributed Work*, P. Hinds and S. Kiesler, Eds. Cambridge, MA: MIT Press, 2002, pp. 259–282.

[32] M. L. Markus, B. Manville, and E. C. Agres, "What makes a virtual organization work?," *Sloan Management Review*, vol. 42, pp. 13–26, 2000.

[33] M. Michlmayr, "Managing Volunteer Activity in Free Software Projects," presented at Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track, 2004.

[34] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "A case study of Open Source Software development: The Apache server," presented at Proceedings of the International Conference on Software Engineering (ICSE'2000), 2000.

[35] B. A. Nejmeh, "Internet: A strategic tool for the software enterprise," *Communications of the ACM*, vol. 37, pp. 23–27, 1994.

[36] M. O'Leary, W. J. Orlikowski, and J. Yates, "Distributed work over the centuries: Trust and control in the Hudson's Bay Company, 1670–1826," in *Distributed Work*, P. Hinds and S. Kiesler, Eds. Cambridge, MA: MIT Press, 2002, pp. 27–54.

[37] R. J. Ocker and J. Fjermestad, "High versus low performing virtual design teams: A preliminary analysis of communication," presented at Proceedings of the 33rd Hawaii International Conference on System Sciences (HICSS-33), 2000.

[38] E. S. Raymond, "The cathedral and the bazaar," *First Monday*, vol. 3, 1998.

[39] E. S. Raymond, "The Cathedral and the Bazaar," *Knowledge, Technology & Policy*, vol. 12, pp. 23–49, 1999.

[40] S. Sawyer and P. J. Guinan, "Software development: Processes and performance," *IBM Systems Journal*, vol. 37, pp. 552–568, 1998.

[41] W. Scacchi, "The software infrastructure for a distributed software factory," *Software Engineering Journal*, vol. 6, pp. 355–369, 1991.

[42] W. Scacchi, "Understanding the requirements for developing Open Source Software systems," *IEE Proceedings Software*, vol. 149, pp. 24–39, 2002.

[43] C. B. Seaman and V. R. Basili, "Communication and Organization in Software Development: An Empirical Study," Institute for Advanced Computer Studies, University of Maryland, College Park, MD, USA 1997.

[44] I. Stamelos, L. Angelis, A. Oikonomou, and G. L. Bleris, "Code quality analysis in open source software development," *Information Systems Journal*, vol. 12, pp. 43–60, 2002.

[45] K. J. Stewart and T. Ammeter, "An exploratory study of factors influencing the level of vitality and popularity of open source projects," presented at Proceedings of the Twenty-Third International Conference on Information Systems, 2002.

[46] P. C. van Fenema, "Coordination and control of globally distributed software projects," in *Erasmus Research Institute of Management*. Rotterdam, The Netherlands: Erasmus University, 2002, pp. 572.

[47] D. B. Walz, J. J. Elam, and B. Curtis, "Inside a software design team: knowledge acquisition, sharing, and integration," *Communications of the ACM*, vol. 36, pp. 63–77, 1993.

[48] M. B. Watson-Manheim, K. M. Chudoba, and K. Crowston, "Discontinuities and continuities: A new way to understand virtual work," *Information, Technology and People*, vol. 15, pp. 191–209, 2002.

[49] P. Wayner, *Free For All*. New York: HarperCollins, 2000.

[50] Y. Yamauchi, M. Yokozawa, T. Shinohara, and T. Ishida, "Collaboration with lean media: How open-source software succeeds," presented at Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW'00), Philadelphia, PA, 2000.

[51] R. K. Yin, *Case study research: Design and methods*. Beverly Hills, CA: Sage, 1984.

9. Tables

Table 1. Projects selected for analysis.

	EGroupWare	Gaim	Compiere
Development Status	5 - Production/Stable	5 - Production/Stable	5 - Production/Stable
Programming language	PHP	C	Java
License	GNU General Public License (GPL)	GNU General Public License (GPL)	Mozilla Public License 1.1 (MPL)
Developer count	42	12	44

Table 2. Coding scheme for task assignment mechanism

Name	**Code**	**Description**	**Example**
Relationship of Task Assignment	DD	Developer assigns task to developer	
	DU	Developer assigns task to user	
	UD	User assigns task to developer	
	UU	User assigns task to user	
Task Assignment Mechanism	SA	Assigning tasks to self	`I'd like to work on this part.` `If you'd like, I'll make the package.`
	ACP	Assigning tasks to a specific person	`Luke, could you check this bug?`
	AUP	A assigning tasks to an unspecified person	`Can someone please do a brief test, replacing config.php with newconfig.php?`
	AO	Assigning task to a person who is neither a developer or user	`I'll ask one of my friends if he can come up with quality sounds as well.`
	SCO	Suggest consulting with a certain person to do the task	`You'd better ask Jorg and work with him to solve it.`

Table 3. Frequency of destinations of task assignment by project.

Task Assignment Mechanisms	Code	Frequency		
		EGW (%)	Gaim (%)	Compiere (%)
Self assignment	SA	37(52.9)	60 (59.4)	16 (57.1)
Assign to a specified person	ACP	15(21.4)	18 (17.8)	9 (32.1)
Assign to an unspecified person	AUP	12(17.1)	22 (21.8)	1 (3.6)
Ask an outsider (a person not in the project development team)	AO	0	1 (1.0)	0
Suggest consulting with others	SCO	6 (8.6)	0	2 (7.2)
Total of Task Assignment Messages		70 (100)	101 (100)	28 (100)

Table 4. Comparison of destination of task assignment by developers and users.

Task assignment mechanism		EGW		Gaim		Compiere	
		Devel. (%)	Users (%)	Devel. (%)	Users (%)	Devel. (%)	Users (%)
	Code	Dx	Ux	Dx	Ux	Dx	Ux
Self assignment	SA	25 (51.0)	12 (57.1)	28 (51.9)	32 (68.1)	8 (44.4)	8 (80.0)
Assign to a specified developer	ACP-xD	1 (2.05)	2 (9.5)	3 (5.6)	5 (10.6)	3 (16.7)	1 (10.0)
Assign to a specified user	ACP-xU	12 (24.5)	0 (0.0)	8 (14.8)	2 (4.3)	5 (27.7)	0
Assign to an unspecified person	AUP	7 (14.3)	5 (23.8)	15 (27.7)	7 (14.9)	0	1 (10.0)
Ask an outsider	AO	0	0	0	1 (2.1)	0	0
Suggest consulting with other developer	SCO-xD	3 (6.1)	2 (9.5)	0	0	0	0
Suggest consulting with another user	SCO-xU	1 (2.05)	0	0	0	1 (5.6)	0
Suggest consulting with outsider	SCO	0	0	0	0	1 (5.6)	0
Total of Task Assignment Messages		**49 (100)**	**21 (100)**	**54 (100)**	**47 (100)**	**18 (100)**	**10 (100)**

Note: Codes for columns are combined with the codes for the rows to form a complete code; e.g., xD + Dx yields code DD.