COLLABORATION THROUGH SUPERPOSITION:

HOW THE IT ARTIFACT AS AN OBJECT OF COLLABORATION AFFORDS TECHNICAL INTERDEPENDENCE WITHOUT ORGANIZATIONAL INTERDEPENDENCE

James Howison
Carnegie Mellon University
jhowison@cs.cmu.edu

Kevin Crowston
Syracuse University
crowston@syr.edu

ABSTRACT

This paper develops a theory of collaboration through superposition: the process of depositing separate layers on top of each other over time. The theory is developed in a study of development of community-based Free and Open Source Software (FLOSS), through a research arc of discovery (participant observation), replication (two archival case studies) and formalization (a model of developer choices). The theory explains two key findings: 1) the overwhelming majority of work is accomplished with only a single programmer working on a task and 2) when tasks appear too large for an individual they are more likely to be deferred until they are easier, rather than being undertaken through structured teamwork. It is theorized that this way of organizing is key to successful open collaboration where the IT artifact is the object of collaboration, because it allows the co-production of technically interdependent artifacts through motivationally interdependent work. The affordances of software as an object of collaboration are used as a framework to analyze efforts to learn from FLOSS in other domains of work and in the IS function of for-profit organizations.

This research was partially supported by the National Science Foundation under Grants 04–14468, 05-27457 and 07–08437.

COLLABORATION THROUGH SUPERPOSITION:

HOW THE IT ARTIFACT AS AN OBJECT OF COLLABORATION AFFORDS TECHNICAL INTERDEPENDENCE WITHOUT ORGANIZATIONAL INTERDEPENDENCE

James Howison
Carnegie Mellon University
jhowison@cs.cmu.edu

Kevin Crowston
Syracuse University
crowston@syr.edu

INTRODUCTION

New ways of organizing closely associated with information systems, such as Open Source Software and Wikipedia, are surprising because they blend two circumstances that the literature has consistently found to be challenging: working at a distance (e.g., Lipnack and Stamps, 1997; Olson and Olson, 2000) and working with sporadically available volunteers (e.g., Dunlop, 1990; Handy, 1988). Unsurprisingly, then, many researchers and managers look to this way of organizing for inspiration, hoping to learn from their example (e.g., Agerfalk and Fitzgerald, 2008; von Krogh and von Hippel, 2006; Scacchi et al., 2006; Stewart and Gosain, 2006).

The relationship between information technology and organization has been of great interest to Information Systems (e.g., Crowston and Malone, 1988; Desanctis and Poole, 1994; Markus and Robey, 1988; Orlikowski, 1992). In these new ways of organizing information technology plays two important roles. First, IT plays a relatively well-studied role as a medium of collaboration (e.g., Daft and Lengel, 1986; Dennis et al., 2008). Second, these forms have at their centre an IT artifact in a novel role as the *object* of collaboration, such as software source code or wiki pages.

This paper develops the empirically grounded theory that the affordances of the IT artifact as an object of collaboration are tightly bound to the success of these novel ways of organizing.

A strong line of literature has argued the structures of technical interdependence embedded in the work itself determine the type of organization most suited to that work (e.g., Malone and Crowston, 1994; Thompson, 1967; Van de Ven et al., 1976). Recently, however, this line of thinking has been challenged by studies driven by a practice view of work (Shea and Guzzo, 1987; Wageman, 1995; Wageman and Gordon, 2005). The challenge suggests that similar work can be accomplished successfully with very different patterns of interdependence, and moreover that patterns of appropriate task interdependence are driven as much by factors such as individual preferences and technological affordances as by unchangeable requirements of work.

This paper introduces the impact of participant motivations on patterns of interdependence and work. It argues that the motivational environment is important even when building software collaboratively, a task where the technical interdependence of the work has been held to be especially determinative (e.g., Herbsleb et al., 2001). In short, the paper argues that the IT artifact as an object of collaboration affords collaboration through superposition: the laying down of motivationally-independent layers over time, each layer taking previous layers as its starting point and, in turn, providing a base and inspiration for the next.

As an empirical example and source of ideas for theorizing, this paper examines Free (Libre) and Open Source Software development, herein abbreviated as FLOSS. FLOSS is a canonical type of distributed, online production (e.g., von Hippel and von Krogh, 2003; Wasko and Faraj, 2005). The projects studied in this paper are community-based FLOSS projects, meaning that they have no institutional existence (e.g. non-profit foundations) nor do they have significant corporate

involvement; in this way the projects chosen epitomize what is most novel in the FLOSS phenomenon; its least hybridized form.

Existing research on FLOSS development has concentrated in three areas: inputs (including motivations), processes (such as coordination or governance) and outputs (such as implementations or software quality). It is established that FLOSS participants are driven by a variety of motivations (e.g., Feller et al., 2005; Ghosh et al., 2002; Lakhani and Wolf, 2003; Lakhani and von Hippel, 2003; Roberts et al., 2006), including the need for software itself, learning, ideological commitment and, at least for long-term participants in well-known projects, reputation, although perhaps not to the degree suggested by economists (e.g., Lerner and Tirole, 2002). While there is a growing contingent of participants who participate as part of a paid job (Roberts et al., 2006), in the community-based projects that this paper focuses on participants are participating in their "spare time" (Luthiger and Jungwirth, 2007). Process research has focused on coordination, documenting the prevalence of practices such as self-assignment of tasks (the essence of volunteerism), "short-cut" decision making processes as well as a tendency to "code first" and then work together on integration (Crowston et al., 2005; Yamauchi et al., 2000). Comparisons of coding practices found that FLOSS projects tend to have smaller and more frequent check-ins than commercial projects (Mockus et al., 2002).

Research has also focused on characteristics of these projects which may help to overcome the seeming limitations of the organizational form, including control (Gallivan, 2001), governance (O'Mahony and Ferraro, 2007), ideology (Stewart and Gosain, 2006), past collaborative ties (e.g., Hahn et al., 2008) and knowledge flow between FLOSS projects (e.g., Daniel and Diamant, 2008). Literature examining the outputs of FLOSS projects includes research studying software

quality (e.g., Schach et al., 2003) and work establishing a definition for IS success in the FLOSS context (e.g., Crowston et al., 2006).

Unfortunately there are very few studies which link across these areas, linking motivations, organization and success. There are a small number of articles that draw on the job design tradition (e.g., Hertel, 2007; Ke and Zhang, 2008) and work contributed by academics with strong practitioner backgrounds in FLOSS that emphasizes the volunteer environment as fundamental to the organization of successful FLOSS production (Capiluppi and Michlmayr, 2007; Michlmayr, 2004). Attention has also been drawn to the possible connection between technical structures in the artifact and the volunteer context, arguing that more modular structures ought to attract more volunteers (Baldwin and Clark, 2006; Conley and Sproull, 2009; MacCormack et al., 2006). This paper contributes to existing literature on FLOSS by linking motivation and process, arguing that by looking at these together one can understand the FLOSS phenomenon, and the contingencies in its adaptability, better.

The purpose and structure of this paper

The overall purpose of this paper is empirically grounded and illustrated theory development (e.g., Weick, 1989, 1995), undertaken though an unfolding arc of *discovery*, *replication* and *formalization*. *Discovery* consisted of four years of participant observation in a community-based FLOSS project, *replication* consisted of an archive-based field study in two similar FLOSS projects. Finally the theory is *formalized* through a rational expectations model of developer decision-making. The first section of the paper, therefore, is divided into three parts, one for each part of this arc. The discussion that follows is in two parts: firstly we examine the specific role of the IT artifact (Benbasat and Zmud, 2003; Orlikowski and Iacono, 2001) in supporting this model of organizing. We focus on the affordances of IT artifacts as the object of work and their

interaction with the resource environment faced by these production communities. Secondly, drawing on this discussion, we demonstrate the usefulness of our theory by examining the challenges of adapting FLOSS organization for other types of work and institutional environments, including the IS function in traditional, for-profit businesses.

DISCOVERY: PARTICIPANT OBSERVATION

For over four years the first author participated in and observed the BibDesk project, a community-based FLOSS project producing a reference manager akin to Endnote. This section is written in the first person from the perspective of the first author, highlighting the epistemological origins of the understandings it presents. Participant observation began with the sensitizing concepts of motivation and interdependency. Data collection was through journaling and periodic review of archival records. Simultaneously, I was actively reading the growing FLOSS literature as well as testing and expressing my growing insights through writing and presenting in both academic and practitioner venues (such as O'Reilly OSCON). In this way the insights were shaped by the contrast between the understandings of FLOSS I encountered and my experience in BibDesk.

Into the field

I let my case emerge naturally from my day-to-day practice as an academic, adopting FLOSS tools wherever possible, from Word to LaTeX, Endnote to BibDesk, SPSS to the R statistics package. BibDesk supported my writing work well. The project has always been open source, founded by a graduate student at UCSD; many of the participants were fellow graduate students and all were volunteers and none met face to face, making BibDesk a good non-hybridized case of community open source. Within its niche BibDesk is a successful project: it has consistently

been in the very top percentile of active Sourceforge projects and, as of October 2008, listed 13 developers, although only 5 were consistently active throughout my observation period, the others entering and leaving over time.

The working life of BibDesk occurs in a number of different communication venues: the deeper one's participation, the more venues one encounters. My first encounter with the project was through the application itself; I consider the experience of using the application to be an important shared foundation for communication within a project. The project mailing lists were the next venues I encountered, both the bibdesk-users list and the bibdesk-dev list. My first message to the list suggested a feature improvement; in reply the founder gently and encouragingly directed me to a specific section of the code.

Fired up, I was able to download the code through the anonymous CVS provided by SourceForge and attempted to build the project from source. This introduced me to another project venue, the source code repository. When I first downloaded the code ("checked out") and attempted to compile the application, the code did not successfully build. This was, I introspected, a frustrating experience, immediately undermining my motivation to contribute. Girded, however, by commitment to the project as a research setting, I determined that the source of the errors was that the default build settings placed external libraries used by the project in a different place than the compilation settings now used by all the developers.

My first contribution, therefore, was a Perl script to download these libraries to the correct place on the disk, then check out the BibDesk source and build the application. I submitted this script to the developer mailing list where it was well received: the developers hadn't known that their source didn't build easily on potential contributors machines. This task was also my first introduction to the project's unit of contribution: the patch. A patch is a set of changes, a "diff"

which is applied to the codebase adding new functionality or fixing bugs. As developers work on the code, they share that work with other developers by submitting patches. These patches can be sent to the mailing list, but are more commonly used to update a shared copy of the source code kept in a source code control system (in the case of BibDesk, a system called CVS), called "making a commit". The resulting process is somewhat like co-authors sharing Word documents with tracked changes. Figure 1 shows the relationship between four different views of the project's source code.

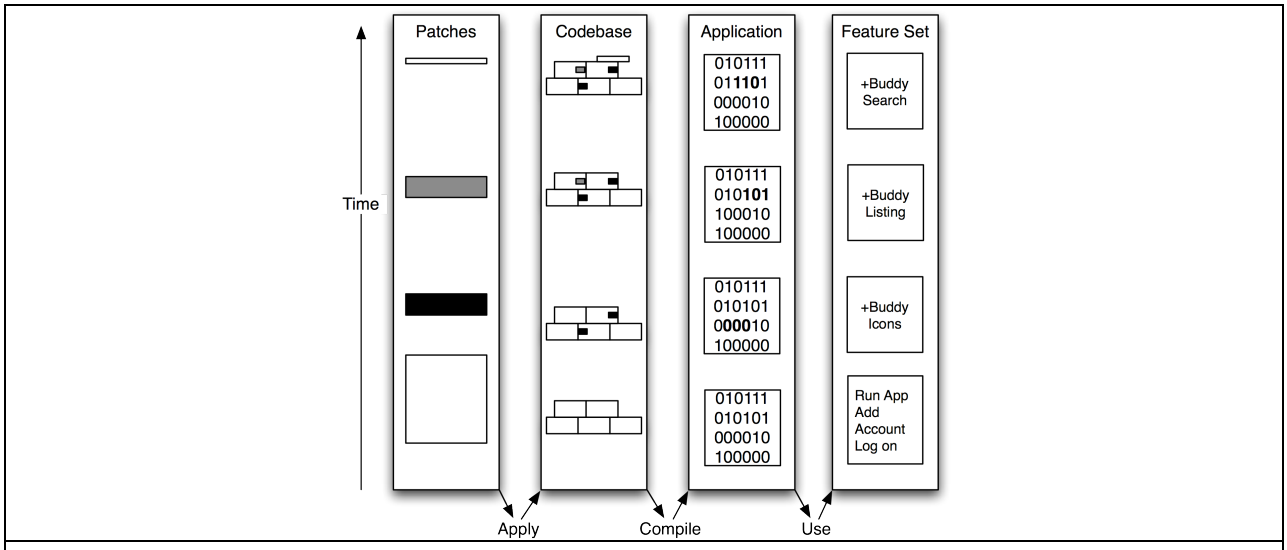

Figure 1: Four views of project code. Patches build up over time through superposition, changing the codebase and application, implementing features.

With longer participation I also came to encounter another important venues: trackers. These are web-forum like issue management systems. BibDesk used just the default types from Sourceforge: Bugs and Request for Enhancements (new features). While most discussion remained on the mailing list, trackers were used as longer-term memory especially when mailing list threads became fractured by long pauses in a discussion.

As I became more involved in the project I found that my understanding of the life of the project was not organized by any technological feature of any of these tools, such as threads or tracker

items, but episodes of work in which the developers and users were engaged, which I call tasks. Tasks provide coherence to work but leave traces scattered throughout different venues. Thus a task might begin with messages on a mailing list, then posts in a tracker, then as a patch, then as a CVS check-in and finally as a functional change to the application itself. Others tasks might simply show up in CVS then the application.

Three vignettes

In this section I outline three episodes of work in the BibDesk project, designed to highlight the findings of my participant observation.

Container Column One task I undertook as part of the BibDesk project was to create a new kind of column in the summary display labeled “Container”. This column displays the Journal title for articles and the title of the conference or proceedings for Conference Proceedings (journal and conference title are separate fields in a BibTeX record and so would otherwise be displayed in two columns). I undertook this task in April 2005 motivated purely out of personal annoyance at the small screen of my laptop, which made it difficult to see both columns at once.

I worked on this task in private, without sharing my plans with the project beforehand, since I thought I had a good understanding of what I wanted to achieve and did not want to bother the other developers with simple questions about the code, especially if I wasn’t able to complete the task. My first exposure of the idea was an email to the developer's list, describing the feature and including a patch. I hadn't committed it to CVS—even though I had commit privileges—because the patch didn't work quite as I'd hoped (it wouldn't sort properly), but worked well enough to show my intentions. The project founder reviewed the patch, endorsing the intended change, and replied with comments on how to fix the sorting issue. I found this motivating (especially as it

was an embarrassingly simple error on my behalf) and so after 4 hours further work, I fixed the sorting and committed the patch, thus making the results of my work available to other developers and users. In total I estimate that it took about 20 hours of work spread over three days.

This episode was fairly typical of my involvement, and, by observation, the patterns of involvement of other developers. Tasks tended to be primarily undertaken by an individual programmer in a relatively short period of time at the developer's own behest, motivation and timing. Support between developers, if there was any, was unplanned; more a case of reaching out in case anyone was there than a case of planned inter-dependency. Tasks result in a single patch which bundles up the changes necessary to effect the changes to the application, resulting in immediately useful incremental progress.

Bibdesk 2.0: The second illustrative episode is in strong contrast to the style of task related above. The BibDesk 2.0 episode was a long running period in which the intention of the group was to release a re-factored and largely rewritten BibDesk. Yet as of June 2010 the current version was 1.5.2, which is to say that BibDesk 2.0 never emerged. The effort began in the same way as BibDesk itself: as a private project of the project founder that was eventually moved into BibDesk's public repository. One of the main intentions was to move BibDesk from a very heavily BibTeX-centered project to a generic reference manager able to be integrated with document preparation systems other than LaTeX, a persistent user request. BibTeX was to be replaced as the underlying file format and instead be simply one export format among many. In addition it was the stated intention that the work would make contribution easier by refactoring the codebase.

The 2.0 project, however, never caught hold. I managed to build it a number of times, but the functionality was low compared to the 1.0 version and none of the developers could adopt it for day-to-day work. Rather than switching work to the BibDesk 2.0 version the participants—other than the project founder—largely continued to tweak BibDesk 1.0. It was not the case that there was significant tension about this, but the reality was that it was hard for the rest of the project to change tack and focus on BibDesk 2.0, even though the participants generally agreed with a need for a re-write and did contribute some small testing and work on the BibDesk 2.0 module (which remains in the BibDesk code repository). Instead work progressed on the BibDesk 1.0 module in small steps. Nonetheless the developers eventually achieved most of the features planned for 2.0: a vastly improved group system, a very flexible non-BibTeX template system, the ability to store more than one file per entry and to use file aliases instead of full paths. The project achieved this through small additions over time, retaining the basic architecture of the 1.0 software and even the reliance on the BibTeX file format.

The BibDesk 2.0 episode envisaged a fairly radical reconfiguration of the relationships between the developers. It placed other developers as dependent on the completion of work by the project founder or, had others joined the effort, interdependent on contributions by each other where the payoff in working software was weeks if not months down the track. This is quite different than the normal working mode of incremental small steps with immediate payoffs. Even though the group took a consensus decision to hold off adding new features to the 1.0 codebase, as time stretched forward this was tacitly abandoned; the developers returning to their non-interdependent incremental development process.

2003 email (on discussing a desirable feature): *I really want to use this, but **the conditions have never quite been right** - either I was waiting for … RSS+RDF (now looks like it'll never happen) or … an XML bibliographic file format … (could happen now, **but I ran out of free time**).* **2007** email (on checking in working code for this feature): *It was **much easier than I expected** it to be because the existing groups code (and search groups code) was very easy to extend. Kudos - **I wouldn't have tried it if so much hadn't already been solved well**.*
Table 1: An episode from BibDesk, discussing the ability to subscribe to online reference lists. These two emails were sent four years apart: the complex work was deferred until other work, done for its own purposes, had made the original desired feature possible to accomplish through short, individual work.

Web groups: The third vignette shows that this incremental, layered process is surprisingly capable. Table 1 provides an example of this style of development reflected in two emails from the project founder, written four years apart. The first email is a response to a suggestion I made regarding a feature to subscribe to publication lists on academic's personal homepages. The project founder had previously conceived of this and agreed that it would be a useful feature but never began work on it. I also found the task too complicated for the time I was able to devote to the project. The task was thus left languishing. Four years later, somewhat out of the blue, the project founder (by then relatively inactive) contributed the feature, emphasizing that in the intervening years the task had become "much easier". This was, he explained, because of the incremental layered work of other developers; work undertaken not in preparation for Web Groups but for other features that literally just happened to also support Web Groups. This work had prepared the ground, so that a developer working alone in a matter of days could complete the feature that earlier had been too much work to complete.

Participant observation findings

These vignettes illustrate the major findings of the participant observation.

Organization and Interdependency: The unit of contribution in the project was the patch, wrapping up code changes associated with a particular task. Project members built on each other's work, but just as a patch alters only what is there already, they very rarely relied on each other's future availability or planned work. They did not work entirely alone, but sought and gave support only spontaneously. Tasks tended to be relatively short, on the order of a few days of work. Work that did not fit this model was difficult to complete, sometimes failing completely. At other times, such work was deferred, usually revisited only when other independent work had, in an unplanned way, changed the codebase so that the work could be accomplished in short independent tasks.

Motivation and Organization: While motivation is an often-studied topic in research on FLOSS, it has only been studied through surveys and interviews. Participant observation affords the addition of introspection. My experience pointed to the involvement of two aspects of volunteer motivation that fit with the organization of work above. Firstly I was only able to work on the project in my free time and my free time did not come consistently, therefore I was not keen to take on tasks that I did not feel I could finish in the time I knew I had available. Secondly, I worked alone because I did not want to rely on the free time and commitment of others to finish my work; I felt I had no right to request their time and since their commitment was also unpredictable I did not want to rely on their portion of shared work being completed. I did not want to be left "high and dry" if they, quite legitimately, had to attend to their real life.

Collaboration through Superposition: The identification of the patch as the unit of contribution lead to the conceptualization of superposition as vital to the way software is produced in the BibDesk project. Work proceeded in small, independent tasks, each with a functional pay-off through its changes to the codebase and thus application (see Figure 1 [p 7]).

These layered on top of each other over time, each creating the circumstances taken as given for the production of the next layer in a way analogous to the superposition of rock strata.

Superposition through layering is a way of understanding software—and its construction over time—that is related to, but distinct from, modularity. As mentioned above, modular architectures are suggested as fundamental to good software and to attracting volunteer participants (Baldwin and Clark, 2006; Conley and Sproull, 2009; MacCormack et al., 2006). A module has as its distinguishing characteristic its separateness from other code, as measured by low coupling, and the manner in which it groups related functionality (Parnas et al., 1981), as measured by high cohesion. By contrast, a software layer, as conceived in this paper, may draw on code from many functional modules to deliver its payoff; its distinguishing characteristic is that it takes as its starting point only what is already there. Indeed most patches seemed to span across formal modules since they were focused on delivering new functionality rather than optimizing code. Modularity may assist with producing software in layers, by reducing the amount of the codebase that needs to be altered and thus understood. But they are not the same thing: modularity is a characteristic of the codebase, while superposition through layering is a characteristic of its production, as shown in Figure 1 [p 7]. This conceptualization is used in the model developed below and elaborated in the Discussion.

REPLICATION: ARCHIVAL CASE STUDIES

The insights reported above are derived from one individual's experience of a single project. If these findings describe a socio-technical phenomenon worth theorizing about they ought to be repeated in similar socio-technical environments. Thus in order to challenge and strengthen the insights gained from participant observation, we undertook an archive-based field study. The

specific goal of the study was to see whether the findings from the participant observation, specifically the layering of short, individual episodes of work and the deferral of complex work, could be replicated.

Case Selection: Two cases similar to BibDesk and to each other were selected for this analysis: Fire and Gaim, both instant-messaging clients. They are appropriately similar in that they are entirely volunteer-based without revenues or foundations; they are hosted on Sourceforge and use similar tools. They are approximately the same size as Bibdesk and undertake development for applications used by their developers. As well, the two are comparable to each other because they develop similar applications.

Data: Participant observation had indicated that work proceeds across many project venues, and a coherent understanding of the project's organization could not be obtained from single venues, such as the mailing list, alone. Therefore data collection was as comprehensive as possible: from Mailing lists, Source code repositories (CVS and SVN), Forums, Issue Trackers and Release Notes. We analyzed an inter-release period (approximately 45 days) for each project, chosen to be close in calendar time so the projects work would be dealing with similar external contexts.

Analysis: The goal of the analysis was to document how work was done in the two projects. Starting from the participant observation and working inductively, we developed a set of concepts to describe the work as a set of tasks. We defined a *Task Outcome* as a change to the shared outputs of the project, usually the software but also potentially including documentation or the project website. A *Task*, then, was a series of *Actions* undertaken by *Participants* contributing to the *Task Outcome*. *Actions* could be observed in the participant observation study, but in this archival study we relied on Documents, such as emails or CVS check-ins or log

messages, to provide evidence of these Actions. Table 2 [p 15] shows definitions of these concepts.

Concept	**Definition -** *Example*
Document	Archived Content - *An email message, Tracker comment, Release Note*
Event	An Event causes a Document, or many Documents, to be archived *Sending an email, releasing a version*
Participant	A distinct individual involved with the project *Larry Wall (the person), Sean Egan (the person)*
Identifier	A string identifying a Participant *Larry Wall (the name), larry@wall.org (an email address)*
Task Outcome	A change to the shared output of the project *A new feature, a fixed bug, updating documentation*
Action	Work which contributes to a Task Outcome *Writing a translation, requesting a feature, writing code*
Task	The sequence of Actions contributing to a particular Task Outcome *Creating a new "buddy search"*
Table 2: Concepts used in organizing archival records	

In this framework, the goal of the data analysis was to assemble evidence of the *Tasks* performed in each project—in terms of *Participants*, *Actions* and *Outcomes*—from the evidence in the collection of *Documents* for each project. We did this by organizing the archive into collections of *Documents* for each *Task*. We began with the Release Notes, and the README file: literally, a file in the source named README, containing notes from the developers about the code, updated as the code is updated, and often including the participants' own description of *Task Outcomes*. *Documents* from the various data sources relevant to each *Task* were then grouped together. However, the Release Notes and changes to the README file do not necessarily record all completed *Task Outcomes*, so we worked iteratively until we had assigned all the records from the source code repository (since they all necessarily alter the shared work product), creating new *Tasks* as needed. A total of 106 *Tasks* were identified, 62 for Fire and 44 for Gaim, 65 from the Release Notes, 31 from changes to the README file and 10 from changes to the source code repository alone.

Once we had a set of *Tasks*, each described by an outcome and including a collection of *Documents*, the *Documents* were examined to identify the *Actions* contributing to the *Task*

Outcome. *Actions* were classified using the inductively developed coding scheme shown in the Appendix. Actions were also coded for their timing and the Participants involved. **Table 3** shows sample *Tasks*, with their *Actions* and the codes applied to them. The top cell shows a task in which multiple programmers worked; this is, perhaps, a common image of collaboration. The bottom cell shows contrasting examples of when only a single programmer worked on a task.

Co-Production			
Date/Gap	**Actor (overall role)**	**Action**	**Code Applied**
		Gaim Task 2: manual browser security fix	
20 July 2002	kareemy (user)	reports bug	Use Info. Provision
1D 5h 50m	lschiere (dev)	attempts diagnosis	Code Info. Provision
(undated)	robot101 (p dev)	writes patch	Code Production
20D 9h 41m	seanegan (dev)	checks in patch	Review
10D 18h 10m	seanegan (dev)	tweaks fix	Polishing Prod.
1D 20h 8m	chipx86	re-writes fix	Core Production
1D 3h 20m	seanegan (dev)	move fix to branch	Management Work

Solo Production			
Date/Gap	**Actor (overall role)**	**Action**	**Code Applied**
		Fire Task 57: user list duplicate fix	
06 Dec 2002	gbooker (dev)	fixes bug	Core Production
		Gaim Task 3: iconv library integrated	
02 Aug 2002	seanegan (dev)	adds library	Core Production
19m 52s	seanegan (dev)	edits ChangeLog	Documenting Work
26m 10s	seanegan (dev)	integrates library	Core Production
		Fire Task 5: scroll on PgUp	
19 Nov 2002	nkocharh (p dev)	makes PgUp scroll	Core Production
		Fire Task 29: AIM buddy icons	
27 Oct 2002	gbooker (dev)	checks in buddy icon code	Core Production
(same time)	gbooker (dev)	changes ChangeLog	Documenting Work
39m 3s	gbooker (dev)	add jpg icons	Polishing
1h 22m	gbooker (dev)	add bitmap icons	Polishing
12h 1m	gbooker (dev)	.buddyicon save	Polishing
1h 22m	gbooker (dev)	add bitmap icons	Polishing
3D 13h 1m	gbooker (dev)	fix IRC icons	Polishing
3D 18h 34m	gbooker (dev)	fix memory leak1	Core Production
1h 6m 23s	gbooker (dev)	fix memory leak2	Core Production

Table 3: Illustrative Tasks. These tables show tasks as re-organized from the project archives, with Actions undertaken by Participants contributing to Task Outcomes. The Actions have been coded according to their contribution to the outcome.

The goal of our analysis was to determine if the findings from the participant observation held in other settings and with a more rigorous analysis. We present evidence from the data concerning two of these findings: the length of and participation in work episodes and deferral of work.

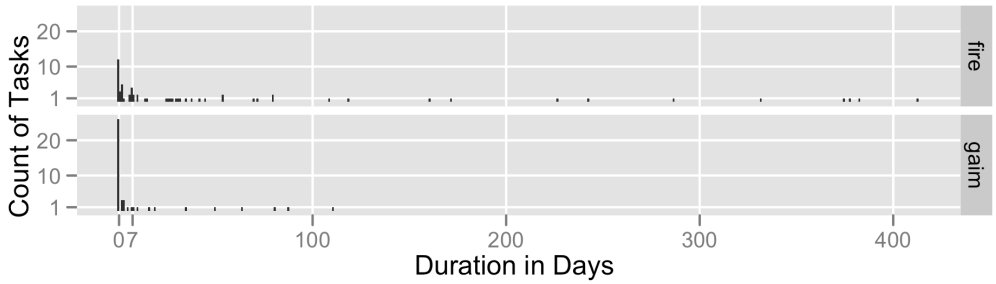

Figure 2: A histogram showing the skewed distribution of the length of tasks.

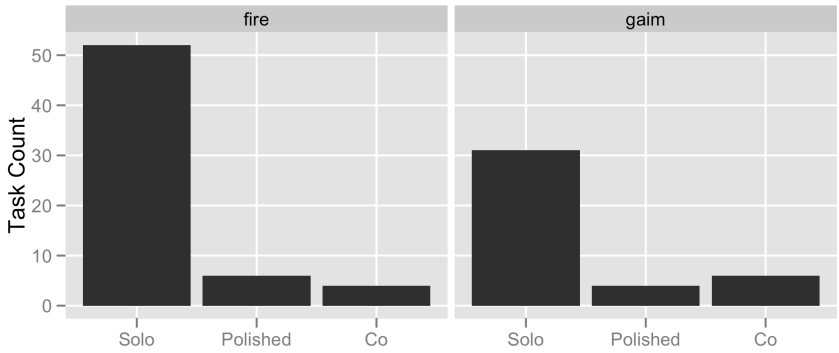

Figure 3: Tasks tended to involve only individual programmers. Figure shows tasks in Fire and Gaim classified primarily by the number of programmers. N=106.

Short and Individual Episodes: We found clear evidence replicating the finding of work being undertaken in short and individual episodes. First, the mean and median duration of a *Task* was shorter than 1 week, as shown in Figure 2; the few longer outliers are discussed below. Second, as shown in Figure 3, overall approximately 80% of the *Tasks* involved only a single participant writing code. A further approximately 10% of *Tasks* were primarily programmed by a single participant, with a small amount of 'polishing' work done by another participant, work such as

fixing a spelling mistake. Less than 10% of *Tasks* involved more than one person programming; these we call co-work. Even within the co-work episodes, only one involved any *Actions* coded as *Management work* to synchronize the work of two programmers. The co-work *Tasks* showed no signs of systematically greater complexity, such as involving more lines of code.

Difficult work was deferred: There was evidence that work was deferred when it seemed hard to complete. Figure 4 [p 19] shows a plot of long running *Tasks*. The release period is marked on the horizontal axis. The early *Actions* in these *Tasks* were all coded as Support, usually feature requests or posts accepting a feature as desirable. Close inspection shows that all the production work for these *Tasks* was completed relatively quickly at the end of the *Task*, during the release period, even on those tasks that had been outstanding for months.

Qualitative investigation of these tasks provides evidence of similar processes of deferral to those found in BibDesk; a feature request was acknowledged as desirable, but the work was initially considered too difficult to undertake. For example, *Task* f_9 in Figure 4 [p 19] consists of a feature request made in March 2003. At that time there is discussion amongst the developers of the desirability of the feature, yet no work is done until October 2003, when the developer comments that an unrelated feature has simplified the request, "*This is possible now with the 'once' option probably I will check it in the next week or so*". In the specific case of these Instant Messaging clients, the addition of protocol specific libraries, written outside the project, facilitated waves of tasks, resolving outstanding acknowledged feature requests or bugs. These tasks are also striking for what did not occur: despite their early endorsement as desirable, there was no evidence of detailed planning, assignment or breakdown of work towards these tasks, nor even explicit anticipation of a new library version. Rather the exogenous arrival of a new library prompted individual short integrative tasks.

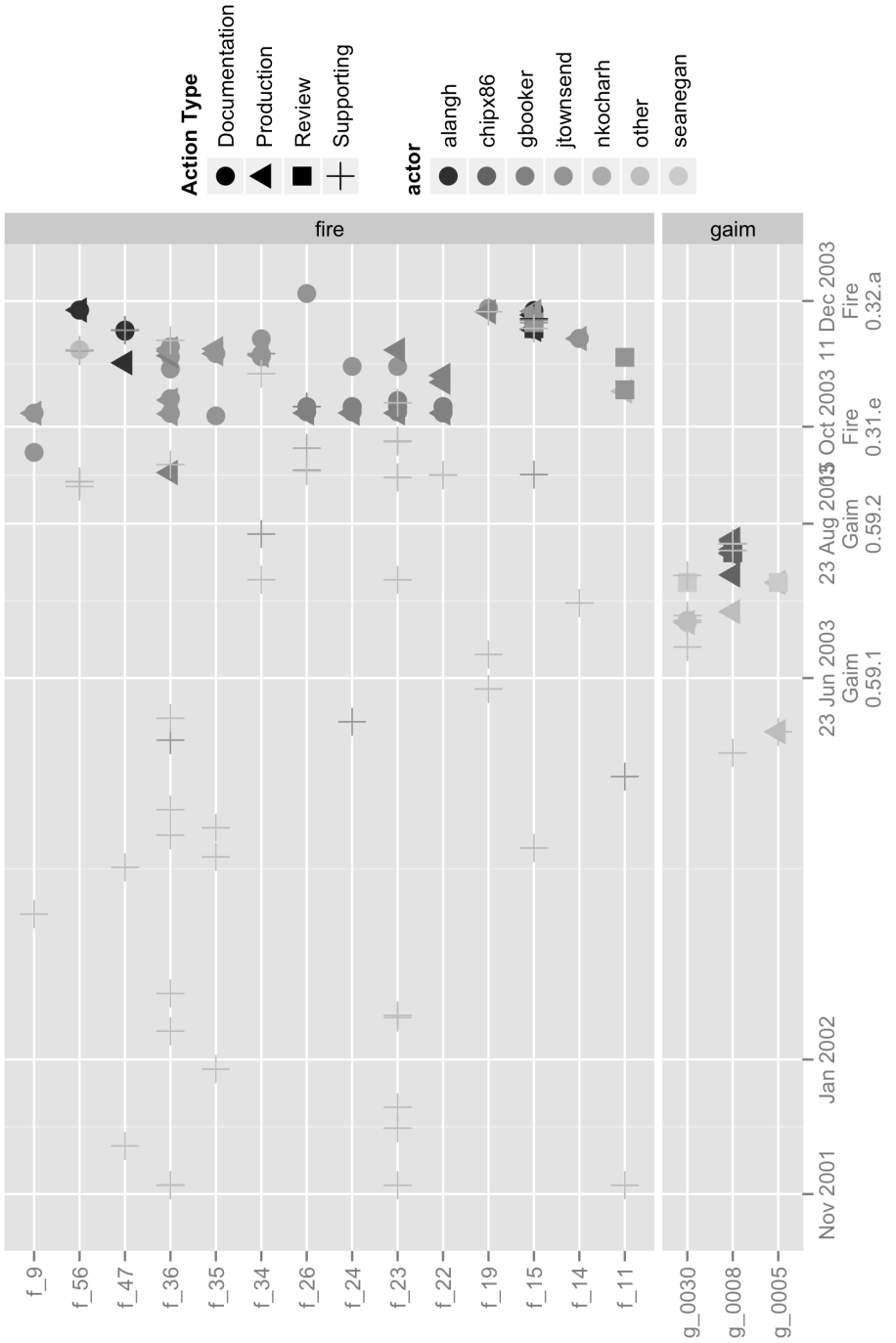

Figure 4: Difficult Tasks were deferred. The figure shows long running Tasks begun by requests by peripheral participants (black circles). These were accepted by core participants (grey plus symbols) but deferred until the inter-release period, when production work (grey triangles) was undertaken in short tasks with only single programmers. n=17 (of 106 total tasks)

Replication Findings

Overall, archival analysis of two additional projects supported the findings from the participant observation: the work in Fire and Gaim was undertaken overwhelmingly through individual, short episodes of work. There was very little evidence of planning work (only found in a single task) and no evidence of resource management. Complex work was deferred, rather than being broken down into smaller components to be undertaken collaboratively.

FORMALIZATION: THEORY ELABORATION

In order to explain these replicated empirical findings, in this section we develop a formal model of developer's decisions about whether to undertake a particular programming task. The model attempts to capture essential features of this decision making through a simplified, stylized model. By model, we mean a logical, formal analysis that draws implications for behavior from the basic principles (Kaplan, 1964). The model to be developed draws on a logical analysis of two concepts: the structure of software and individual decision making. We present our assumptions about each of these aspects in turn.

Dependency in layers

Considering the software structure first, participant observation found that the unit of contribution was the patch and that patches nearly always delivered changes with immediate payoffs in functionality or usability. This structure of development works because of the fundamental characteristic of software discussed above: it is additive, meaning that it can be built up through the superposition of layers over time.

Layering gives rise to two important types of dependencies in software production. Figure 5 [p 21] illustrates these relationships. From top to bottom the layers depend on each other

functionally; without the lower layers the code will not compile and the application simply will not be able to run. In the diagrams to follow all higher layers depend on an unbroken stack of lower layers, with the higher layers said to have a *functional* dependency on the layer beneath it. Figure 5 also shows the second type of dependency, called a *utility* dependency. Some layers of software have no direct utility to the user and depend on higher layers to expose their functionality, thereby releasing value to a user. In these diagrams layers that have direct utility are colored gray, while layers that rely on higher layers for their utility are colored white.

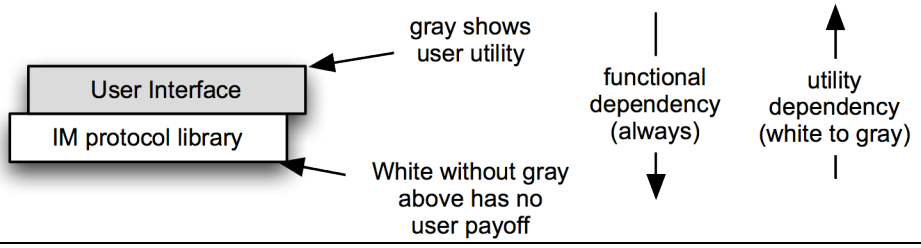

Figure 5: All layers have functional dependencies on those below them. Gray layers have direct utility, and white layers have utility dependencies on gray levels above them.

From the perspective of development these two types of dependency have two implications. The first is that a missing functional dependency removes the utility from layers that depend on it, since without a full stack the software can't run and the "gray" layer cannot deliver its utility to the user. Figure 6 [p 22] shows this situation. The second implication is that there is no restriction that lower layers cannot also have direct utility; it is not the case that all utility must exist in the top layer alone. Figure 7 [p 22] illustrates this situation with two features of an instant messaging client: buddy display and buddy search. Buddy search clearly depends on being able to display buddies because without it search results can't be displayed. Yet buddy display is already useful on its own. Search has a functional dependency on display, but display does not have a utility dependency on search. This is shown in these diagrams by stacking two gray boxes on top of each other.

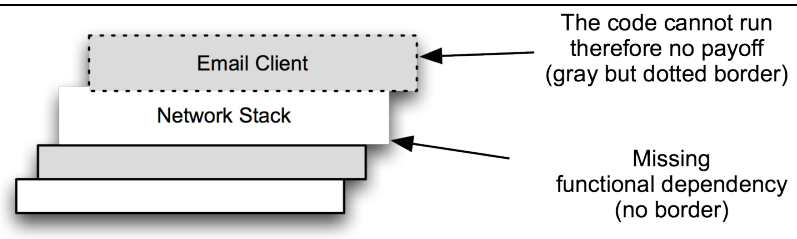

Figure 6: Without a network stack, shown missing as a box without a border, an email client is incapable of delivering its utility. The lack of utility is shown by giving the gray box a dotted line

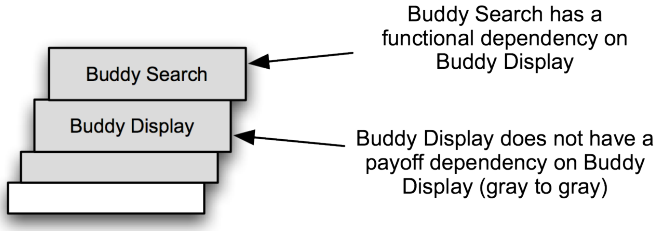

Figure 7: Layers with direct utility can also be built upon, so that there is a functional dependency without a utility dependency, shown with gray on gray.

Decision-making

Second, we consider the process of decision-making. Agents in rational choice models are modeled as making a choice between alternatives, such as choosing which basket of products to buy or which investments to make. The basic rational choice model is simple: agents assess the benefits and the costs of each course of action (Eq. 1). Costs are understood as opportunity costs: the lost benefit of the alternative not chosen. Eq. 2 shows the condition: that the benefits of the choice exceed the benefits of the alternative.

Of course an agent cannot see the future; they can only make their decision on their expectations of both the benefits and the costs of the action. We can therefore restate the decision equation, in terms of the Utility derived from the outcome ($U_{outcome}$,), adjusted for the probability that the decision will lead to the desired outcome (Eq. 3).

(Eq. 1) Benefit > Cost	**(Eq. 2)** $B_{choice} > B_{alternative}$
(Eq. 3) $E(B_{choice}) > E(U_{outcome}) \times E(P(success))$	

To model the impact of judgments about the likelihood of success, we build on the expectancy-valence model of decision-making (Vroom, 1964), a process theory of motivation whose essence is the suggestion that the "attractiveness of a particular task and the energy invested in it will depend a great deal on the extent to which the employee believes its accomplishment will lead to valued outcomes" (Steers et al., 2004). This theory was chosen as it matches well the motivational introspection from participant observation, reported above.

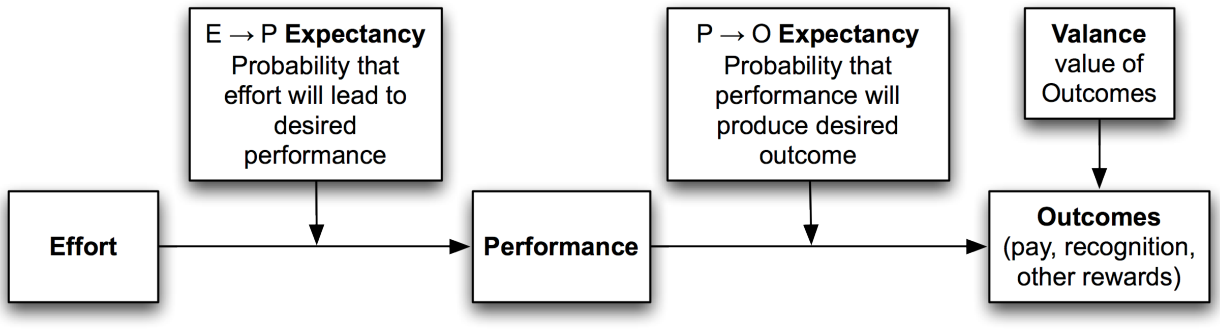

Figure 8: Expectancy-Valence model of motivation. Adapted from (Samson and Daft, 2005, p. 534)

Figure 8 shows that there are two separate expectancies in this theory. The first is Expectancy-Performance (E→P), which argues that the individual calculates the probability that effort will lead to desired performance, *P(e→p)*. The second is Performance-Output (P→O), which argues that the individual also calculates the probability that performance will lead to the desired outcome, *P(p→o)*. The decision condition can thus be restated with all components being expected values:

(Eq. 4) $E(B_{choice}) > E(U_{outcome}) \times E(P(e \rightarrow p)) \times E(P(p \rightarrow o))$

Stylized Facts and Assumptions

For analytical clarity, this section details a set of stylized facts and assumptions that are the basis for the model, summarized in Table 4. These assumptions are based on the FLOSS research literature and participant observation findings described above. Some assumptions will be relaxed after the initial analysis.

Table 4: Stylized Facts and Assumptions
Bullets show assumptions which will be relaxed; right column shows justification

Participants only work for a utility payoff		Bounded Rationality
Participants are good, but not perfect judges of task difficulty		Bounded Rationality
Participants know the limitations of their judgment		Bounded Rationality
Participants only know their free time a short period in advance		Part. Observation.
Contributions are always shared under an open source license		Part. Observation
Participants only derive utility from their own use of the software	•	Analytical Clarity
All participants have the same set of skills and free time	•	Analytical Clarity
There are no exogenous sources of code or solutions	•	Analytical Clarity

The model considers agents who are potential developers on a FLOSS project. As found in participant observation, agents are assumed to have a limited amount of spare time, which is outside their normal course of life—things such as paid work, family life, etc. It is assumed that these agents use the software regularly outside their spare time, perhaps for work or study. This regular use allows the agent to see opportunities to improve the software (and thereby the effectiveness of the rest of their activities). Initially we assume this is their sole motivation. The improvement in effectiveness is therefore the value of $U_{outcome}$ and is known to the agent.

Time in the model is divided in turns. The choice facing the agent in each turn is constrained to be binary: they either choose to spend the turn attempting to contribute to a FLOSS project or they choose not to. What they do with their spare time otherwise is immaterial, but this activity is assumed to have a low but certain payoff.

The model envisages agents making two estimates related to evaluation of the choice of coding as an activity. First, they have to estimate whether their effort will result in the needed performance, i.e., estimating $P(e \rightarrow p)$. The agents set their expectation of $P(e \rightarrow p)$ based on their assessment of the chance that they will be able to complete the task within the time available. To make this estimate, agents inspect the current codebase and make their best guess of the amount of time it would take compared against the time they have available. For example, if the time they had available was three hours and they just need to fix a spelling mistake, then their estimate of $P(e \rightarrow p)$ would be very high. Contrariwise, if they needed to design a new spell checking algorithm in that three hours, then the estimate of $P(e \rightarrow p)$ would be very low.

Having agents make this assessment requires further assumptions. Firstly, they are assumed to be good, but not perfect, judges of their skill level. Secondly, agents are also assumed to be good, but not perfect, judges of the complexity of the tasks, and are assumed to be good but not perfect at understanding the current codebase. The agents are assumed to know their limitations. Together these assumptions create a small chance of failure every time a developer undertakes a task. At the stage of comparing complexity to time available, agents are assumed to know their time availability for the length of the turn, but not beyond that. Agents are assumed not to rely on possible future availability. Additionally, all agents are assumed to have the same level of skill. This assumption will be relaxed below.

The second assessment that the agents make is to assess $P(p \rightarrow o)$, an assessment of the probability that accomplishing the task set forth will enable them to use the application in such a way as to unlock the utility that motivated them and achieve the expected rise in productivity. For the case of individual work this is set to, and is expected to be, 1 (i.e. certain). This is an assumption that the agent is able to use the application as anticipated.

It is also assumed, at this stage, that there is no source of exogenous code or solutions; the development effort of the developers is the only source of advancement for the project. This rules out outsourcing through payment or the discovery of libraries of code from outside the project. This assumption is an obvious simplification and will be relaxed below.

Finally, we assume that the agents all share their patches under an open source license. Why such revealing takes place is outside the scope of this model. However, the use of an open source license has three implications that together allow participants to build on other's work confidently: 1) derivative works are allowed, 2) no royalties must be paid and 3) contributions are not revocable; the contributor cannot withdraw them. Together these factors mean that even if a developer were to regret the decision to contribute, their contribution would remain freely available and therefore agents do not have to expect continued cooperation from others. This allows the superposition of new work on old to result in technical interdependence without organizational interdependence.

Model analysis

With these assumptions and background, it is possible to consider an individual agent making a code or no-code decision. On each turn, each agent consults their current set of motivating opportunities and the current codebase to assess whether they can, given their skills and available time, undertake a development task, making their best assessment of the easiest way to change the codebase to achieve the desired outcome. If they estimate that the expected benefits of the change outweigh the opportunity costs they begin work. If they successfully complete the work, the result is available to all the other developers in the next turn. Participants return to the codebase from time to time and reconsider work they previously rejected or failed to complete.

Given a set of agents with varying time available and tasks with differing utility payoffs, we consider implications for work in the project.

Figure 9: As development proceeds over time through superposition, functional dependencies become backwards only dependencies. They depend only on what is already present.

Individual, layered work can proceed: The simplest situation is that a participant inspects the codebase and their set of desired features and finds a task which is sufficiently motivating, but also within their abilities given the time known to be available to them. Such a situation is shown in Figure 9, where a layer with utility (gray) is layered atop an existing layer with utility (also gray). These situations can be called "backwards-only" dependencies: all that is relied on are the results of actions already completed and certain: the current codebase and its permanent availability. The FLOSS license assumption provides important safeguards for the payoff of such work. If necessary layers could be removed in the future then the new code would lack its functional dependencies and be unable to provide the utility desired by the developer. This situation describes the case of a developer patching a proprietary piece of software—if and when the software or its source availability changes, the patch can become unusable.

The "missing step" problem: A second possible situation is a participant inspecting the current code base and their desires and judging that the work needed to implement a desired feature is greater than their skills will allow them to accomplish in the time they know they have available.

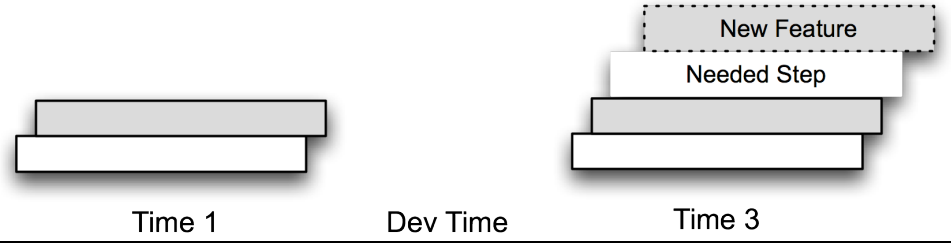

Figure 10: A missing functional dependency becomes a forwards and backwards dependency.

Figure 10 illustrates this situation with two separate layers, the desired `New Feature` and an additional `Needed Step`, representing the increased amount of work needed to achieve the goal. An individual agent under the assumptions given above will not try to implement both layers at once as they expect to fail to finish in the time available, in the decision equation E($P(e \rightarrow p)$) will be so low that not coding will always provide a higher expected return, even for very large values of $U_{outcome}$. Nor would it be rational for an agent to work on either layer in isolation because without the white `Needed Step` layer, the payoff of the gray `New Feature` layer is not available (a missing functional dependency) and without the potential to finish the `New Feature` in the time available the `Needed Step` will not be built (since it is missing a utility dependency). Since we assume that all participants have the same skills, they face the same situation: the work will not be undertaken.

This situation reflects a fundamental dilemma in collaboration. It is a stylized and contextualized way of restating a core problem of collective action: if the complexity of work is beyond the individual capabilities of participants then some way to mitigate this must be found. The following section models two solutions to this situation: collaboration through coordinating multiple actors, which is well known, and productive deferral, which is believed to be novel.

Interdependent collaboration: A natural way to resolve this situation is to have agents communicate and potentially make agreements between themselves. Agents may discover other

participants who are also motivated by the `New Feature` in Figure 10 [p 28]. Together they can assess that if both of them work they can accomplish the work; one works on `Needed Step` and the other on `New Feature`, as depicted in Figure 11. The capacity to work together with both receiving the payoff opens the way to resolving the "missing step" dilemma.

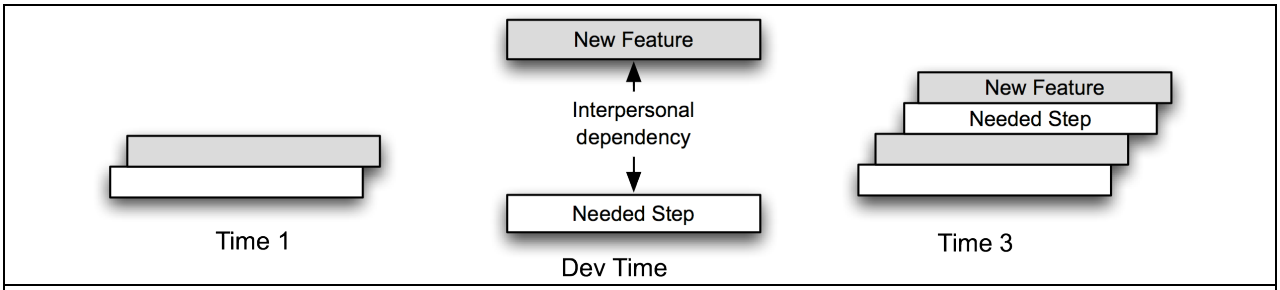

Figure 11: The missing functional dependency can be added if two developers work concurrently, although the risk of non-completion rises due to partner failure and coordination costs.

While this seems a reasonable solution even under the assumptions in place in this model such collaboration introduces new risks that could undermine motivation and reduce volunteers' participation in projects. First, collaboration introduces a new source of non-completion risk because any collaborator also has a chance of failure, their $P(e \rightarrow p)$ is less than 1, and therefore may not complete their part of the agreement, rendering the joint payoff unavailable (either because a functional or a payoff dependency is unsatisfied). This situation can be incorporated into the model as a change in the expectation of the second part of the risk term, $E(P(p \rightarrow o))$, to represent the risk that the expected outcome will not eventuate, regardless of the agent's success in initial performance. We represent this risk simply by setting each agent's $E(P(e \rightarrow p))$ to their collaborator's $E(P(p \rightarrow o))$. For example, if the individual likelihood of success for each developer is 0.8, the overall chance of success for the two components combined is only 0.64, meaning that the overall utility of the work would need to be substantially higher for the collaboration to seem worthwhile.

In addition, concurrent development introduces another well-known problem: the two layers have to be designed to work with each other. In the model so far this integration has been assumed to be trivial because developers have only been building on finished code that can be easily inspected for fit. However, concurrent work is not available for inspection because it is being simultaneously developed. This creates a usability coordination cost, which we model as increased risk of non-completion (for either participant), decreasing *P(e→p)*. Furthermore, these costs are known to participants—and known to affect their partner—and are therefore transferred to the expectation of the *P(p→o)* term, as above. We represent the probability of avoiding misfitting work as θ. In this way each agent's *P(p→o)* term becomes dependent on their collaborator, making each agent's decision condition:

(Eq. 5) $E(B_{choice}) > E(U_{outcome}) \times E(P(e \rightarrow p)_{self} \times \theta) \times E(P(e \rightarrow p)_{other} \times \theta)$

Since both agents have *P(e→p)* < 1and $\theta < 1$, co-work will always be more risky than work conducted through individual steps with immediate utility payoffs. This models for two participants only, but there is nothing stopping larger numbers of participants agreeing to take on even more complicated tasks together. However these sort of agreements are exponentially less likely as the failure of any single individual undermines the payoff for all, and super-linearly increases risk through extra coordination effort which is now between three or more simultaneously developing components, rather than two.

This formal statement of the collaboration challenge explains the findings above of large amounts of individual work: it is simply less risky and thus of higher expected payoff for the participants. At the same time, some work will be perceived to be so worthwhile that attempting co-work makes rational sense.

Deferring difficult work: The above solution to the collaboration dilemma is well known. However, a second possibility identified in the empirical work is for the developer to defer work for the current turn, and wait to see how the codebase changes over time. It is possible that through the independently motivated and executed work of others the desired `New Feature` will become possible to implement with only individual work within the agent's known free time. In short the `Needed Step` simply appears and turns the collaborative dilemma into a relatively simple backwards-only dependency.

What trickery is this? Above we argued that participants motivated by instrumental payoffs would never build `Needed Step`, a layer without a utility payoff. It can't come from an exogenous source, because we are currently assuming that these do not exist. How then does `Needed Step` emerge?

The answer has to do with the extraordinary flexibility of software and the situated nature of the developer's cognition. Initially a task may seem to require work that is otherwise valueless (since it has a utility dependency on as yet uncompleted work), but as other work—perhaps just individual backwards-only work—changes the software over time, another way to build `New Feature` may become apparent. Importantly, the changes would have been made for their own sake, not as part of an interdependent plan to eventually build `New` Feature. This situation was encountered by the first author in participant observation and depicted in Table 1 [p 11], where the founder of BibDesk initially perceived the desirable task as too much work but considering it from time to time until code—written for an entirely different reason—made the task easier enough to undertake through relatively simple, quick and individual work.

In other words, the `Needed Step`/`New Feature` dilemma is not necessarily a hard fact—not a "structural requirement" of a task (Wageman, 1995)—but the result of a developer's cognition.

And the cognition of developers—their estimation of the work needed to build `New Feature`—is highly situated: it responds to the codebase as it is now, not to all possible configurations of code. As the codebase changes, new and often surprising ways to accomplish tasks emerge. In this way, as depicted in Figure 12, potentially problematic interpersonal dependencies requiring trust and communication can be converted to two backwards-only dependencies, each of which can be accomplished through individual work alone. This transformation is central to layered collaboration.

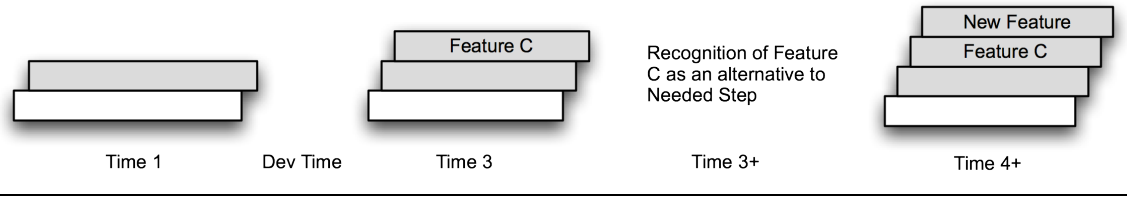

Figure 12: A missing step dilemma can become two separate backwards-only dependencies, if the missing functional dependency proves to have an alternative which itself has a payoff. Note the elongated and indeterminate time scale.

Such "productive deferral" can be incorporated into the rational choice model. Deferred work will be delayed, perhaps forever, which we model by applying a discount to the expected utility of the feature. Note however that agents are not choosing to defer the work as an alternative to both working and not-working; deferral does not take any time, so the benefits of not working or of work on a different feature are still available. Note also that deferral is also not a commitment on the behalf of the agent to undertake the work in future. In this way one can model the decision as between (a) working uncertainly now and likely failing and (b) doing something else now and possibly working more easily later. In some quite conceivable circumstances such deferral would be rationally preferred to collaboration: the combined effects of uncertainty and delay offset by the immediate benefit of doing something else only need be lower than the multiplicative combination of collaboration risks shown above.

There is no need to over-state this point: it is enough that deferral can and does happen. Certainly it does not always happen and for some types of tasks it undoubtedly never happens. Even when it might there is no way to estimate before the fact when the deferred task might eventually be accomplished. Deferring work is far from a replacement for collaboration: it is at best slow and from an traditional management perspective might be totally ineffective, especially in circumstances where it is vital that features be implemented in a timely manner (as we discuss below). Nonetheless, deferral is a novel solution to the collaboration dilemma; it is one that is in keeping with the motivational environment of FLOSS development and the affordances of the IT artifact as an object of collaboration.

Model Extensions

This model presented above is useful but simple; a number of extensions are immediately apparent. Here we consider just three, designed to consider the impact of three empirical aspects of FLOSS discussed in the literature: firstly, participants in FLOSS projects have different areas of expertise and skill/productivity levels; secondly, FLOSS projects can make use of exogenous sources of code; and thirdly, participants are known to be motivated by more than the simple instrumentality of the software.

Unequal skill distribution: First, the model above assumes that all participants have the same skills and productivity; this assumption can be relaxed by modeling skill as a distribution. The immediate impact of this change is to allow the existence of developers for whom the implementation of `Needed Step` and `New Feature` (from Figure 10 [p 28]) can be done in a single step. For example, the developer might have experience of `Needed Step`, e.g., through work on another program, and therefore able to implement it quickly, or simply have

superior skills and productivity sufficient to complete both of the steps in the free time they know they have available to them.

Relaxing this assumption leads to a more realistic model where some tasks are hard for some developers but easy for others. Indeed it is often said that programming is conspicuous for order of magnitude variance in programmer productivity (e.g., Brooks, 1975). The work of highly productive coders could make leaps on which other developers can build smaller but still useful contributions. A diversity of skill levels also changes the calculation of risk in collaboration, since one developer might have less chance of failure than the other and others might be relatively happier to co-work with them.

Under this assumption, a project can speed through some `Needed Step` dilemmas, provided a developer exists for whom the task is comparably easy *and* for whom it provides sufficient utility. This observation points to the importance of continual recruitment, above and beyond added generic effort: by widening the diversity of contributors the project has more chance of solving larger and more complicated tasks even if contributors cannot contribute large amounts of time. Having uncommonly productive members might also assist the project by making deferral more effective for other members by keeping the codebase changing in ways that might make complex work easier. For such reasons it might be appropriate for the project to be more concerned about attracting new, skilled participants than organizing in ways that make most efficient use of the spare time of existing participants, a trade off similar to that observed in listserv-based online communities (Butler, 2001).

Exogenous change: FLOSS projects do not exist on their own; rather they are part of an ecosystem of software and other projects. Although differing licenses reduce the possibility of code movement between some projects, developers have a great deal of software at their

disposal. This is especially true of libraries, which are packaged and documented for re-use and can provide significant functional services. By relaxing the assumption that no solutions or code are exogenously available it is possible for individual developers to overcome the `Needed Step` dilemma by recognizing a library as providing the step, adapting it and pursuing `New Feature` in their known free time. The developer, therefore, is assessing not just the current codebase but also the part of the whole ecosystem of (license compatible) FLOSS code known to them. Further, libraries are designed to be integrated and provide "bundles" of potential features. As found in the study of Gaim and Fire above, their addition can spark new rounds of creativity amongst developers, perhaps enabling tasks that were too complex earlier and therefore deferred or which were not even conceived until the addition of the library.

Reputation and Learning motivations: Finally, the literature on FLOSS motivations is clear: instrumental motivations (for the software itself) are important but are not the only drivers of activity. Two other motivations stand out: reputation and learning, as key examples of instrumental and experiential motivations, respectively.

The importance of reputation is perhaps stronger in the conceptual literature than in the empirical literature, but is nonetheless important because it could motivate developers to take on tasks without certainty of completion or immediate payoff. If the project carefully assigns credit, individuals can increase their reputations by taking on tasks that are known to be collectively valuable but which are not otherwise motivating. An example of this situation might be tasks such as managing infrastructure or undertaking the infrastructural work represented by `Needed Step` above. Having developers motivated by reputation enables projects to proceed through `Needed Step` blockages without resorting to work with direct interpersonal interdependencies, still working only through layered individual work.

Yet reputation seems likely to be important for co-work as well, since reputation can be understood as revealed quality and commitment. Since participants can see the public successes of other's efforts (and are relatively shielded from their private failures) they may be more willing to trust others and enter into agreements co-work. On the other hand, reputation takes time to build, so is unlikely to be operative in the very early stages of a project. Only once the project has established itself as well-known and valuable can reputation have value outside the project itself (Roberts et al., 2006).

Learning is part of a set of experiential motivations that also includes sheer enjoyment of the. Intrinsic motivations are significant because they change the types of payoffs permitted, allowing participants to implement steps that were previously ruled out because they did not provide immediate functional payoffs. This effect has the helpful result of turning a step with functionality but no immediate instrumental utility (a 'white' box) into step where the payoff is, for example, the learning the developer anticipates during their attempt (a 'gray' box). Such alternative payoffs allow experientially motivated developers to choose to implement gaps, such as `Needed Step`, only because they will learn from the activity, regardless of whether they, or someone else, ever implements the `New Feature` that has a functional dependency on the `Needed Step`.

Furthermore experiential motivations mean that even ultimately unsuccessful work has value to the developer, since one can learn from failure, or simply enjoy the process. This reduces the perceived risk for the developer: one might say, "hey, I'll give it a try and even if it doesn't work out I'll at least have learnt something". Given also the ability to communicate, and thereby get opportunistic coaching from an interested and skilled community, coding to learn appears to be a

very powerful motivator; indeed Larry Wall, the founder of the Perl project calls open source development, "learning in public".

Such motivations could be incorporated in the model by adding (not multiplying) a new term to Eq. 4 on the benefit side, reflecting the expected utility of the experience itself, $E(U_{experience})$. Since it is additive this term is unaffected by the risk of failure or any of the multiplicative risks of collaboration and will make work more likely to be undertaken. Successful work done for experiential reasons can bridge motivational gaps in the project and such work can provide layers that make the work of other participants much easier, supporting a whole new raft of backwards-only tasks. This analysis helps to clarify why experiential motivations are so important to volunteer work—they make it possible to motivate contributors to work on tasks from which they derive no instrumental benefits.

DISCUSSION

The theory developed in this paper emphasizes how much can be accomplished without risky interdependent collaboration (particularly when intrinsic motivations, such as learning, are considered) and describes a novel solution to the collaboration problem: productive deferral. The model of organizing presented in this paper is entwined with affordances of IT artifacts, four of which are highlighted below. Understanding these better improves analysis of the challenges in adapting FLOSS organizing for other work, including the traditional IS function.

Collaboration affordances of the IT artifact

The theory presented in this paper rests in part on the affordances of software as a type of IT artifact. This argument is similar in form to other theories of IT affordances and organizing (e.g.,

Dennis et al., 2008), but novel in that it emphasizes the IT artifact as an object of work, rather than as a communication medium.

Layerability: As described above, software has the affordance that it can be built in relatively small layers and additions can be integrated as they arrive. Further, layers can have independent payoffs, even for very small additions (such as adding a search function). The ability to build complex, technically interdependent work in immediately motivating layers over time is very important to the theory presented in this paper. However, it is clear that much work is not like this. For example an airplane certainly can be built in steps: first the fuselage, then the engines, and finally the wings. But until it is complete none of these steps provide any utility payoff. Small steps added to an existing base, each with their own sufficient instrumental payoff (forming "stackable incentives"[1]), seems to be particular to informated work (in the sense of Zuboff, 1989).

Low instantiation costs: By instantiation costs we mean the costs of moving from a design to a useful artifact (and not the process of creating the design). For software this is moving from software code to a running application and is very cheap, simply requiring a computer and a compiler. For other work, however, such as building a house, this can be very expensive. While one could conceive of a community collectively altering and layering a digital design for a house (or car, see below), the fact that there is a comparatively high cost of applying such alterations to the finished artifact means that the use value of the changes is very hard to realize, undermining the types of motivation seen in FLOSS projects. This characteristic is very important for collaboration through superposition; if it is expensive to rebuild the existing work to place your

[1] We are indebted to an interlocutor at a conference for this pithy phrase. Unfortunately, despite many attempts, we have been unable to identify them to thank them by name.

new layer upon, then adding that layer is itself very expensive. Adding a room onto an already functional house does not duplicate the entire house.

Distribution costs: The Internet has drastically reduced distribution costs for software, which is usually delivered via the network at no marginal bandwidth cost for the user. Low distribution cost allows small layers to spread out to the community and provide the basis for other's work. Prior to Internet software delivery, updates involved printing CDs (or copying tapes) and shipping them to customers, a much more expensive proposition. Extremely low instantiation and distribution costs are very important to the model of collaboration presented in this paper and are a characteristic of IT artifacts as an object of collaboration.

Rewindability: Thirdly software is rewindable: as participants add layers they do not commit the entire project to retaining them forever (even while giving up the right to remove them unilaterally). Changes can be undone, especially when source code management systems like CVS are used. Even when another developer commits a change to a shared source code tree, it is possible for others to decide whether they will include those changes in their compiled application. This characteristic is quite unlike the great majority of work in which actions and their impacts are hard or impossible to reverse, in non-informated production (summarized in the advice to "measure twice and cut once") and in services, such as financial advice or customer service. If actions are not easily reversible, trust in other contributors becomes a much more important issue. Gallivan (2001), a meta-analysis of FLOSS case studies, highlighted as surprising the finding that trust was not a commonly discussed element of FLOSS organization, suggesting that control must be playing the missing role. The model presented in this paper, however, suggests that with a rewindable and non-revocable IT artifact as the object of

collaboration, neither trust nor control is as important for collaboration as has hitherto been the case.

Very low financial costs: Fourthly, FLOSS projects are able to exist without special financial investment, resisting the deadline pressures that necessarily accompany it. Community-based software projects are able to take advantage of a set of collaboration tools, from email to download hosting to bug trackers and versioned software repositories, which are almost always free software. As such the costs to set up a project, host it and to make it available for others to discover and build upon are so low as to be irrelevant. While service providers such as Sourceforge do play an important role in facilitating this, and do bear bandwidth costs offset by advertising and demonstration of their technological platforms, many FLOSS projects are able to host their own infrastructure, using excess capacity in personal or business connections.

Low costs means that projects can begin without investing money, can take on new participants without marginal financial costs and can persist indefinitely. This means that the future of the project is not contingent on the sustained provision of financing. Investment—even non-profit oriented investment—has opportunity costs, especially the time-cost of money which entails deadline pressures. When deadlines are important then the tactic of productive deferral is of reduced usefulness and there may be no alternative to working with interpersonal dependencies and therefore bearing the risks and costs outlined above. In the long run, however, slow moving but sustainable projects may out-perform those relying on interdependency.

Challenges for Adaptation

Much of the interest in FLOSS and its development stems from the difficulties encountered in IS development, even when co-located, together with difficulties encountered in distributed work

more generally, especially when crossing organizational boundaries. FLOSS seems to solve these two difficulties by combining them and do so without needing financial investment; a truly remarkable achievement, raising the hope that many useful lessons for conventional development can be extracted from the FLOSS model of organizing (Agerfalk and Fitzgerald, 2008; von Krogh and von Hippel, 2006). Yet the theory presented in this paper, particularly the fundamental role of affordances of IT artifacts, suggests limits and contingencies to this adaptability. The remainder of this section demonstrates the usefulness of the theory developed in this paper by examining attempted adaptations.

Adaptation of a layered model of development seems likely to be most successful in areas that have similar technological affordances. **Wikipedia** was inspired by FLOSS development; at its center is an IT artifact—the wiki—that can effectively be built in layers and where contributions are non-revocable and strongly rewindable. However, instantiation and distribution costs are higher for Wikipedia, despite its informated nature. These costs are higher because the database is stored centrally and each use is its own episode of instantiation and distribution, requiring central server time and bandwidth. In this way a contribution to Wikipedia requires continual maintenance costs if it is to deliver any use value. Bearing these financial costs is a key function of the Wikipedia Foundation and is funded by substantial philanthropy.

Open Hardware is a label applied to efforts to draw on techniques from FLOSS to build hardware, rather than software. The aim of the Open Hardware movement, in projects such as the Simputer and OScar (Open Source Car), is to radically lower the cost of hardware and to radically increase the speed of innovation. These projects have not yet achieved remarkable successes. The primary impediment is that hardware has high instantiation and distribution costs. A circuit board must be designed before it is printed and transported to where it is needed; all

this must happen before, not during, the integration process for any new feature or bug fix. For this reason Open Hardware projects have generally proceeded in two ways: The first is focusing on the design stage, sharing schematics and other documents. However these are not directly useful, so there is a clear unsatisfied utility dependency. The second is to take advantage of the increasing informating of hardware through downloadable firmware or hardware that is more like software, such as Field Programmable Gate Arrays. This strategy has been more effective because it essentially turns hardware into software, but the strategy is not always available.

By far the most hoped for adaptation of the FLOSS form of organizing is for productive hybridization with **the IS function of for-profit enterprises**. This hope has taken two main forms. The first is sometimes known as "inner source" (e.g., Dinkelacker et al., 2002), where a firm attempts to generate an open source community within its corporate boundaries, examples include HP and the US DoD's forge.mil. The second is to integrate FLOSS components into the firm's IS strategy, both for internal IS and for the production of IS for sale (Agerfalk and Fitzgerald, 2008).

The Inner Source strategy presents significant adaptation challenges. It is relatively simple to replicate the IT infrastructure of FLOSS inside a corporation; indeed selling such systems was part of the business models of Sourceforge and Collab.Net. Yet unsurprisingly given the history of IS scholarship (e.g., Desanctis and Poole, 1994; Orlikowski, 1992), simple importation of technology is not sufficient to replicate a socio-technical phenomenon and Inner Source struggles with two key issues.

The first issue is that individual instrumental motivations are de-emphasized since corporate groups usually build for audiences outside the development team, including external customers. Experiential motivations, such as learning and fun, are undermined since individuals rarely

decide the direction of their own work. This leaves Inner Source efforts in much the same motivational and payoff quandary faced by internal knowledge management systems (e.g., Bock, 2005; Kankanhalli, 2005).

The second issue is that firms, due to upfront investment, inherently face deadlines, undermining the usefulness of productive deferral. In these circumstances Inner Source seems most likely to work in two circumstances: those able to sustain a "free time" culture of exploration and learning and those that have significant non-marketed infrastructure needs which meeting can save sufficient money to pay developers directly. In this second case, however, limiting the community to just those inside the corporation does not seem necessary. Extending such communities beyond the boundaries of the firm has seen success in IBM's founding and extension of the Eclipse community.

Other than Inner Source, a second strategy is for firms to adapt FLOSS production into their IS function. The success of this strategy depends in part on the extent to which a firm can be a relatively passive consumer of the project, or whether their strategy requires them to actively develop the code (e.g., Agerfalk and Fitzgerald, 2008; Fitzgerald, 2006; Shah, 2006). The first case is problematic only to the extent of ensuring that the FLOSS licenses match corporate strategies; the theory in this paper does not speak to this issue and clearly many organizations are able to find appropriate matches and use FLOSS software.

If, however, the corporate strategy requires active influence over the project and its codebase there can be significant challenges. Clearly corporate collaboration with FLOSS projects is possible; in fact there are many highly successful examples, such as IBM working with the Apache Foundation, replacing their internally developed web server (WebSphere) with Apache's httpd. Yet the theory articulated in this paper helps to clarify the issues that need to be resolved.

The theory speaks most clearly to whether or not the firm can afford to "fit in" with small layers and deferral of complex work or whether their market imperatives generate deadlines and a strategic need for secrecy.

The case of Apple's Safari browser, based on the open source khtml project, illustrates difficulties in this approach. Apple's market strategy called for high secrecy during product development, and market pressures place a strong premium on rapid time to market. Under the LGPL license Apple was within their legal rights to work in secret; they only had to release their modifications once Safari was distributed. Yet secrecy was not the only driver. By taking the work in-house Apple was able to move much more quickly than the khtml project, short circuiting slow processes of layering and deferral, solving complexity through presumably high-trust, face-to-face interdependent work within their internal programming teams. When Apple released Safari they did indeed release their source code modifications, and announced a desire to work with the khtml community in future, thus sharing on-going maintenance and development costs. Yet the members of the khtml project were displeased, as illustrated by the quotation from a khtml developer in Table 5. Apple's the modifications were too large and had branched from khtml too long ago for them to be easily integrated; their work had not proceeded in observable, short and evolutionary layers.

Do you have any idea how hard it is to be merging between two totally different trees when one of them doesn't have any history? That's the situation KDE is in. We created the khtml-cvs list for Apple, they got CVS accounts for KDE CVS. What did we get? We get periodical code bombs in the form of them releasing WebCore. … They do the very, very minimum required by LGPL. *And you know what? That's their right. They made a conscious decision about not working with KDE developers. All I'm asking for is that all the clueless people stop talking about the cooperation between Safari/Konqueror developers and how great it is. There's absolutely nothing great about it. In fact "it" doesn't exist. Maybe for Apple - at the very least for their marketing people. Clear?*
Table 5: Quotation from khtml developer (http://www.kdedevelopers.org/node/1001)

This story stands in strong contrast to IBM's adoption of Apache's httpd web server. For IBM there was no need for secrecy, since IBM was generating its revenue from services and higher-value added software. Further, the httpd server was already capable enough that IBM was able to fit in with the FLOSS model of working in small, visible steps. The contributions of the IBM developers keep the project active, and may provide volunteer participants with the sorts of "missing steps" that make their work easier. This might even attract additional volunteers, as recently found in a study of corporate impact on the Gnome community (Wagstrom et al., 2010).

Without the active collaboration and support of the community the firm is not able to unlock the promises of "outsourcing to an unknown workforce" (Agerfalk and Fitzgerald, 2008). Firms seeking to drive projects forward by taking complex work in-house, especially in secret, should expect to face similar significant difficulties.

CONCLUSION AND CONTRIBUTION

The theory and empirical work presented in this paper makes useful and significant contributions, albeit not without limitations. The primary limitation is the decision to trade empirical generalizability beyond three specific FLOSS cases from the mid 2000s for the depth needed for theory development. Nonetheless this work is the first to draw together the motivations of participants, the technologies of collaboration and the experience and organization of production into a novel theory with practical implications for research and practice in the fields of Information Systems and Organization Science.

The work makes a contribution to Information Systems because it is a socio-technical theory of organizing where the detailed affordances of IT artifacts play a central role. The theory has implications for the adaptability of FLOSS methods to traditional IS development and for the

interaction of the IS function in organizations with FLOSS communities. A contribution is also made to Organizational Science because a new theory of organizing is described and analyzed, where the task does not structurally determine the appropriate way to organize, but rather appropriate organization is emergent, shaped by the volunteer resource context and flexible technologies of production and collaboration. Finally the concept of "productive deferral" is believed to be novel and may find application in other organizational domains.

This paper began with the observation that the success of FLOSS and other forms of open collaboration is surprising because they face two challenges to organizing: working at a distance and working with volunteers. Working at a distance, outside formal organizations, already sacrifices many traditional sources of control and motivation. Relying on self-motivated volunteers reduces the importance of these; the challenge then is to find a way to organize that draws together relatively independent work into a cohesive and valuable whole, while maintaining a fertile ground for volunteerism. The argument of this paper is that the IT artifact as an object of collaboration affords a solution to this challenge, providing the bedrock on which the superposition of small, self-motivated layers over time can build valuable artifacts and provide mutual inspiration, albeit at the cost of uncertain delay.

Working in this way might be frustrating slow and uncertain from a traditional management perspective that seeks to do more with known, expensive and thus coercible resources. Yet if the challenge is to attract and retain volunteer resources, this way of working makes clear sense. Understanding this is vital to pursuing successful adaptation or hybridization. This way of working is remarkable because the IT artifact—as an object of collaboration—affords not merely doing the same thing faster or more cheaply but a whole new way of collaborating.

REFERENCES

Agerfalk, P. J., and Fitzgerald, B. 2008. "Outsourcing to an Unknown Workforce: Exploring Opensourcing as a Global Sourcing Strategy," *MIS Quarterly* (32:2), pp. 409.

Baldwin, C. Y., and Clark, K. B. 2006. "The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model?," *Management Science* (52:7), pp. 1116–1127.

Benbasat, I., and Zmud, R. W. 2003. "The identity crisis within the IS discipline: Defining and communicating the discipline's core properties," *MIS Quarterly* (27:2), pp. 183-194.

Bock, G. 2005. "Behavioral Intention Formation in Knowledge Sharing: Examining the Roles of Extrinsic Motivators, Social-Psychological Forces, and Organizational Climate," *MIS Quarterly* (29:1), pp. 111.

Brooks, F. P. 1975. "The Mythical Man-Month: Essays on Software Engineering," Addison-Wesley Pub Co., pp. 13–29.

Butler, B. S. 2001. "Membership Size, Communication Activity, and Sustainability: The Internal Dynamics of Networked Social Structures," *Information Systems Research* (12:4), pp. 362.

Capiluppi, A., and Michlmayr, M. 2007. "From the Cathedral to the Bazaar: An Empirical Study of the Lifecycle of Volunteer Community Projects," In *Open Source Development, Adoption and Innovation*IFIP International Federation for Information Processing, J. Feller, B. Fitzgerald, W. Scacchi, and A. Sillitti (eds.), (Vol. 234) Boston, USA: Springer, pp. 31-44.

Conley, C. A., and Sproull, L. 2009. "Easier Said than Done: An Empirical Investigation of Software Design and Quality in Open Source Software Development," In *Proceedings of the 42nd Annual Hawai'i International Conference on System Sciences (HICSS)*.

Crowston, K., Howison, J., and Annabi, H. 2006. "Information systems success in free and open source software development: Theory and measures," *Software Process: Improvement and Practice* (11:2), pp. 123-148.

Crowston, K., and Malone, T. 1988. "Information Technology and Work Organization," In *Handbook of Human-Computer Interaction*, M. Helander (ed.), Elsevier Science Publishers, pp. 1051–1069.

Crowston, K., Wei, K., Li, Q., Eseryel, U. Y., and Howison, J. 2005. "Coordination of Free/Libre Open source software development," In *ICIS 2005. Proceedings of International Conference on Information Systems 2005*Las Vegas, NV.

Daft, R. L., and Lengel, R. H. 1986. "Organizational Information Requirements, Media Richness and Structural Design," *Management Science* (32:5).

Daniel, S. L., and Diamant, E. I. 2008. "Network Effects in OSS Development: The Impact of Users and Developers on Project Performance," In *ICIS 2008 Proceedings*.

Dennis, A. R., Valacich, J. S., and Fuller, R. M. 2008. "Media, Tasks, and Communication

Processes: A Theory of Media Synchronicity," *MIS Quarterly* (32:2).

Desanctis, G., and Poole, M. S. 1994. "Capturing the Complexity in Advanced Technology Use: Adaptive Structuration Theory," *Organization Science* (5), pp. 132.

Dinkelacker, J., Garg, P. K., Miller, R., and Nelson, D. 2002. "Progressive Open Source," In *Proceedings of ICSE '02*Orlando, FL.

Dunlop, J. J. 1990. "Balancing Power: How to Achieve a Better Balance between Staff and Volunteer Influence," *Association Management* (January).

Feller, J., Fitzgerald, B., Hissam, S., and Lakhani, K. 2005. *Perspectives on Free and Open Source Software*, Cambridge, MA: MIT Press.

Fitzgerald, B. 2006. "The transformation of Open Source Software," *MIS Quarterly* (30:4).

Gallivan, M. J. 2001. "Striking a Balance between Trust and Control in a Virtual Organization: A Content Analysis of Open Source Software Case Studies," *Information Systems Journal* (11:4), pp. 277–304.

Ghosh, R. A., Robles, G., and Glott, R. 2002. *Free/Libre and Open Source Software: Survey and Study FLOSS*University of Maastricht: Netherlands: International Institute of Infonomics.

Hahn, J., Moon, J. Y., and Zhang, C. 2008. "Emergence of New Project Teams from Open Source Software Developer Networks: Impact of Prior Collaboration Ties," *Information Systems Research* (19), pp. 369-391.

Handy, C. 1988. *Understanding voluntary organizations*, London: Penguin Books.

Herbsleb, J. D., Mockus, A., Finholt, T. A., and Grinter, R. E. 2001. "An empirical study of global software development: Distance and speed," In *the International Conference on Software Engineering (ICSE 2001)*Toronto, Canada, pp. 81–90.

Hertel, G. 2007. "Motivating job design as a factor in open source governance," *Journal of Management and Governance* (11), pp. 129–137.

von Hippel, E., and von Krogh, G. 2003. "Open Source Software and the `Private-Collective' Innovation Model: Issues for Organization Science.," *Organization Science* (14:2), pp. 209–223.

Kankanhalli, A. 2005. "Contributing Knowledge to Electronic Knowledge Repositories: An Empirical Investigation," *MIS Quarterly* (29:1), pp. 143.

Kaplan, A. 1964. *The Conduct of Inquiry: Methodology for Behavioral Science*, Transaction Publishers.

Ke, W., and Zhang, P. 2008. "Participating in Open Source Software Projects: The Role of Empowerment," In *Proceedings of ICIS08 HCI Workshop*.

von Krogh, G., and von Hippel, E. 2006. "The Promise of Research on Open Source Software," *Management Science* (52:7), pp. 983.

Lakhani, K., and Wolf, R. G. 2003. *Why hackers do what they do: Understanding motivation efforts in Free/F/OSS projects* (Working Paper No. 4425-03), MIT Sloan School of Management.

Lakhani, K., and von Hippel, E. 2003. "How open source software works: "free" user-to-user assistance," *Research Policy* (32), pp. 923–943.

Lerner, J., and Tirole, J. 2002. "Some simple economics of Open Source," *Journal of Industrial Economics* (52:2), pp. 197–234.

Lipnack, J., and Stamps, J. 1997. *Virtual teams: Reaching across space, time and organizations with technology.*, New York, NY: John Wiley and Sons, Inc.

Luthiger, B., and Jungwirth, C. 2007. "Pervasive fun," *First Monday* (12:1).

MacCormack, A., Rusnak, J., and Baldwin, C. Y. 2006. "Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code," *Management Science* (52:7), pp. 1015-1030.

Malone, T., and Crowston, K. 1994. "The interdisciplinary theory of coordination," *ACM Computing Surveys* (26:1), pp. 87–119.

Markus, M. L., and Robey, D. 1988. "Information Technology and Organizational Change: Causal Structure in Theory and Research," *Management Science* (34:5).

Michlmayr, M. 2004. "Managing Volunteer Activity in Free Software Projects," In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track* Boston, USA, pp. 93-102.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. 2002. "Two Case Studies Of Open Source Software Development: Apache And Mozilla," *ACM Transactions on Software Engineering and Methodology* (11:3), pp. 309–346.

O'Mahony, S., and Ferraro, F. 2007. "Governance in Collective Production Communities," *Academy of Management Journal* (50:5), pp. 1106.

Olson, G. M., and Olson, J. S. 2000. "Distance matters," *Human-Computer Interaction* (15), pp. 139–179.

Orlikowski, W. J. 1992. "Learning from Notes: organizational issues in groupware implementation," In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*.

Orlikowski, W. J., and Iacono, C. S. 2001. "Research Commentary: Desperately Seeking the 'IT' in IT Research: A call to theorizing the IT Artifact," *Information Systems Research* (12:2), pp. 121-145.

Parnas, D. L., Clements, P. C., and Weiss, D. M. 1981. "The modular structure of complex systems," *IEEE Transactions on Software Engineering* (11:3), pp. 259-266.

Roberts, J. A., Hann, I., and Slaughter, S. A. 2006. "Understanding the Motivations, Participation, and Performance of Open Source Software Developers: A Longitudinal Study of the Apache Projects," *Management Science* (52:7), pp. 999.

Samson, D., and Daft, R. 2005. *Management - Pacific Rim Second Edition*, Sydney, Australia: Thomson.

Scacchi, W., Feller, J., Fitzgerald, B., Hissam, S., and Lakhani, K. 2006. "Guest Editorial: Understanding Free/Open Source Software Development Processes," *Software Process:*

Improvement and Practice (11:95-105).

Schach, S. R., Jin, B., Wright, D. R., Heller, G. Z., and Offutt, A. J. 2003. "Determining the Distribution of Maintenance Categories: Survey versus Measurement," *Empirical Software Engineering* (8:4), pp. 351-365.

Shah, S. K. 2006. "Motivation, governance, and the viability of hybrid forms in open source software development," *Management Science* (52:7), pp. 1000–1014.

Shea, G. P., and Guzzo, R. A. 1987. "Group effectiveness: what really matters?," *Sloan Management Review* (28:3), pp. 25-31.

Steers, R. M., Mowday, R. T., and Shapiro, D. L. 2004. "The Future of Work Motivation Theory," *Academy of Management Review* (29:3), pp. 379–387.

Stewart, K. J., and Gosain, S. 2006. "The Impact of Ideology on Effectiveness in Open Source Software Development Teams," *MIS Quarterly* (30:2).

Thompson, J. D. 1967. *Organizations in Action: Social Science Bases of Administrative Theory*, New York: McGraw-Hill.

Van de Ven, A. H., Delbecq, A. L., and Koenig, R. 1976. "Determinants of Coordination Modes Within Organizations," *American Sociological Review* (41:2), pp. 332–338.

Vroom, V. H. 1964. *Work and Motivation*, New York, NY: Wiley.

Wageman, R. 1995. "Interdependence and group effectiveness," *Administrative Science Quarterly* (40:1), pp. 145-180.

Wageman, R., and Gordon, F. M. 2005. "As the Twig Is Bent: How Group Values Shape Emergent Task Interdependence in Groups," *Organization Science* (16:6), pp. 687–700.

Wagstrom, P., Herbsleb, J. D., Kraut, R. E., and Mockus, A. 2010. "The Impact of Commercial Organizations on Volunteer Participation in an Online Community," In *Presentation at the OCIS Division, Academy of Management Conference*.

Wasko, M., and Faraj, S. 2005. "Why Should I Share? Examining Social Capital and Knowledge Contribution in Electronic Networks of Practice," *MIS Quarterly* (29:1), pp. p35 - 57.

Weick, K. E. 1989. "Theory construction as disciplined imagination," *Academy of Management Review* (14), pp. 516 –531.

Weick, K. E. 1995. "What Theory is Not, Theorizing Is," *Administrative Science Quarterly* (40:3), pp. 385-390.

Yamauchi, Y., Shinohara, T., and Ishida, T. 2000. "Collaboration with Lean Media: How Open-Source Software Succeeds," In *Proceedings of Computer Support Collaborative Work 2000 (CSCW 2000)*.

Zuboff, S. 1989. *In The Age Of The Smart Machine: The Future Of Work And Power*, Basic Books.

APPENDIX

Coding scheme for Actions, inductively developed	
Code	**Explanation and Example**
Management codes	
Management work	Work done to organize other work. This includes planning, setting deadlines or announcing 'phases' like code/string freezes, assigning or rejecting tasks. This includes re-structuring the infrastructure and declaring bugs fixed, or Patches applied (closing Trackers)
Assigning credit	Thanking people, adjusting the Credits file etc.
Review codes	
Validation work	Validating a coding technique, fix or approach (before or while it is being done)
Review work	Work done to review other work, including checking in code written by others. This includes work that rejects patches etc.
Production codes	
Core production work	Work that directly contributes to the project's outcomes; either through working application code, or through production of user interface elements (logos etc). e.g.: Implementing a feature (not necessarily a check in, since could be checked in on behalf of someone else)
Polishing production work	Smaller changes that polish Core Production contributions e.g.: typos, integrations etc
Documentation codes	
Documentation work	Work that documents the code, application or activities. Includes pointers across Venues (e.g. in a Bug Tracker saying that a Patch has been submitted)
Self-Planning work	Work that documents one's own future activities (planning others' work is Management Work)
Supporting codes	
Use information provision	Providing or seeking information about using the software e.g.: use cases, often RFEs and bug reports.
Code information provision	Providing or seeking suggestions about the code, including how to complete work (code examples or pseudo-code, if it compiles or is a patch against SVN then code Production Work). This includes a developer seeking more information from a peripheral member.
Testing work	Testing application functionality. This includes requesting more information from users in bug reports.