Alone Together: A socio-technical theory of motivation, coordination and collaboration technologies in organizing for free and open source software development

James Howison

Abstract

This dissertation presents evidence that the production of Free and Open Source Software (FLOSS) is far more alone than together; it is far more often individual work done "in company" than it is teamwork. When tasks appear too large for an individual they are more likely to be deferred until they are easier rather than be undertaken through teamwork. This way of organizing is successful because it fits with the motivations of the participants, the nature of software development as a task, and the key technologies of FLOSS collaboration. The empirical findings are important because they ground and motivate a theory that enables a systematic approach to understanding the implications of FLOSS development as a model for adaptation and the future of work. The dissertation presents a process of discovery (participant observation), replication (a systematic study of project archives), and generalization to theory (a model of the rational choices of developers and an analysis of the flexibility of software as a task). The dissertation concludes by enumerating the conditions under which this theory of organizing is likely to be successful, such as non-revokable and rewindable work with incremental incentives. These are used as a framework to analyze efforts to adapt the FLOSS model of organizing for self-organizing, virtual teams in other domains of work.[1]

[1]Future developments of this dissertation, errata and discussion will be available at `http://james.howison.name/pubs/dissertation.html`

ALONE TOGETHER:A SOCIO-TECHNICAL THEORY OF MOTIVATION, COORDINATION AND COLLABORATION TECHNOLOGIES IN ORGANIZING FOR FREE AND OPEN SOURCE SOFTWARE DEVELOPMENT

By

James Howison
B. Economics (Social Sciences) (Hons) University of Sydney 1998

DISSERTATION

Submitted in partial fulfillment of the requirements for the
degree of Doctor of Philosophy in Information Science and Technology
in the Graduate School of Syracuse University

May 2009

Approved_______________________________
Professor Kevin Crowston

Date_______________________________

Contents

Part I

Introduction

Chapter 1

Background and Research Questions

What if you had to organize to not only do something great, but also to attract the very people you need to do those great things, without any money at all? How might that organizing—that way of working—look?

A great deal of literature has studied the more common case of organization in which the future availability of resources is assumed. Teams are studied working, but only after their members are assigned; partnerships between businesses are studied after their members are engaged. Typically this assumption rests on the knowledge that participants are paid for their involvement and, *quid pro quo*, they offer their working time as a resource to be shaped by an active management seeking to improve the overall goal of organizational efficiency.

Increasingly, however, the future of work seems unlikely to look like this. Simply offering money and then telling people what to do isn't always enough to attract the best and brightest, and even if they are attracted it might not be enough to hold onto them or to ensure that they do their best work. Skilled workers, especially in-demand knowledge workers, might be better understood and managed if they were perceived as acting like volunteers (e.g. Handy, 1988; Malone, 2004; Cook, 2008; Barnard et al., 1968; Drucker, 2002).

In some ways it has always been understood that people working together—seemingly without coercion—can do interesting, productive things, but these have been the exception rather than the rule, and have tended to be small-scale and small impact (like managing a local sporting league). Truly hard and important achievements have been accomplished through organizational models with up-front financial investment, strategic planning and just the right amount of incentivized coercion, to ensure both effective and efficient collaboration.

Recently, however, new forms of collaboration have emerged to great acclaim. Phenomena such as free and open source software development and Wikipedia suggest that new technologies are associated with new forms of organizing. Whether these are called "massive virtual collaboration" (Crowston and Fagnot, 2008), "private-collective innovation" (von Hippel and von Krogh, 2003), "wikinomics" (Tapscott and Williams, 2006) or even "crowd-sourcing" (Surowiecki, 2005; Howe, 2006), they suggest that there is something strikingly and importantly novel occurring.

Yet the expectation that truly important things are not done by volunteers in their spare time is so persistent and pervasive, research ought to remain skeptical. This is perhaps doubly-true when claims are made that revolutionary technology lies at the bottom of such transcendence. The rhetorical record is rife with overblown claims about the revolutionary potential of new technologies and attendant claims of its universal and easy application (e.g. Marvin, 1988). Detailed studies reveal the relationship between technologies, organization and effective collaboration to be always much more complex, contingent and suspect (Orlikowski and Barley, 2002; Barley, 1986; Trist and Bamforth, 1951; Orlikowski, 1992).

Nonetheless it is clear that something very interesting is happening around technology supported open collaboration. Artifacts like the Linux operating system and

Wikipedia are threatening long established ways of doing great things in the domains of software and documentation of knowledge. They have emerged from voluntary, non-coerced creative activity and have developed distinctive technologies to support their creation and dissemination. They are worthy of research both for their own sake and for the sake of lessons that might be learnt from them, both for organizational theory and practical application elsewhere.

This dissertation explores these questions and this opportunity and builds an empirically grounded theory of organization in Free (Libre) and Open Source Software (FLOSS[1]) projects[2].

1.1 Research Questions

The purpose of this dissertation is to understand why the FLOSS model of organizing is successful and thereby provide a framework to discuss what can be adapted for

[1]The software produced by these projects are generally available without charge ("free as in beer"). Some (though not all) open source software is also "free software", meaning that derivative works must be made available under the same license terms and therefore can never be made proprietary so they are "free as in speech" thus libre.. There are therefore two similar but distinct communities, the "open source" and the "free software" community. We have chosen to use the acronym FLOSS rather than OSS or even FOSS, to accommodate this range of meanings and respect the freedom (libre) connotation of free. Nevertheless, we have joined the two approaches for this research because their development practices are indistinguishable for the purposes of this dissertation.

[2]The organizational status of, or appropriate analogy for, FLOSS development projects is an interesting research question in its own right. Are they (or are they more like) teams, communities, organizations or groups? Some scholars use team to refer only to the core developers, and community to also include less active participants. Terminology is especially problematic because the notion and extent of the "teamness" of FLOSS production is central to this dissertation. Therefore we have chosen to use the term "project" because it is the term used by participants themselves and leaves the question for empirical data.

other organizational environments. To structure this inquiry, there are three specific research questions, known throughout this dissertation as RQ1, RQ2 and RQ3.

1. How is successful FLOSS production organized?
2. How does this organization interact with the motivations of developers?
3. What are the implications for the adaptation of the FLOSS model of organization in other environments?

This introductory chapter now turns to a more detailed introduction of the theoretical and empirical justification for the proposed research, developing the research problem and describing the intended audiences for the work. The chapter then briefly examines ways to approach research on FLOSS development, making a practical and ethical argument for the methods used throughout this dissertation. The chapter ends with a short summary of the structure of the remainder of the dissertation.

1.2 Research Justification

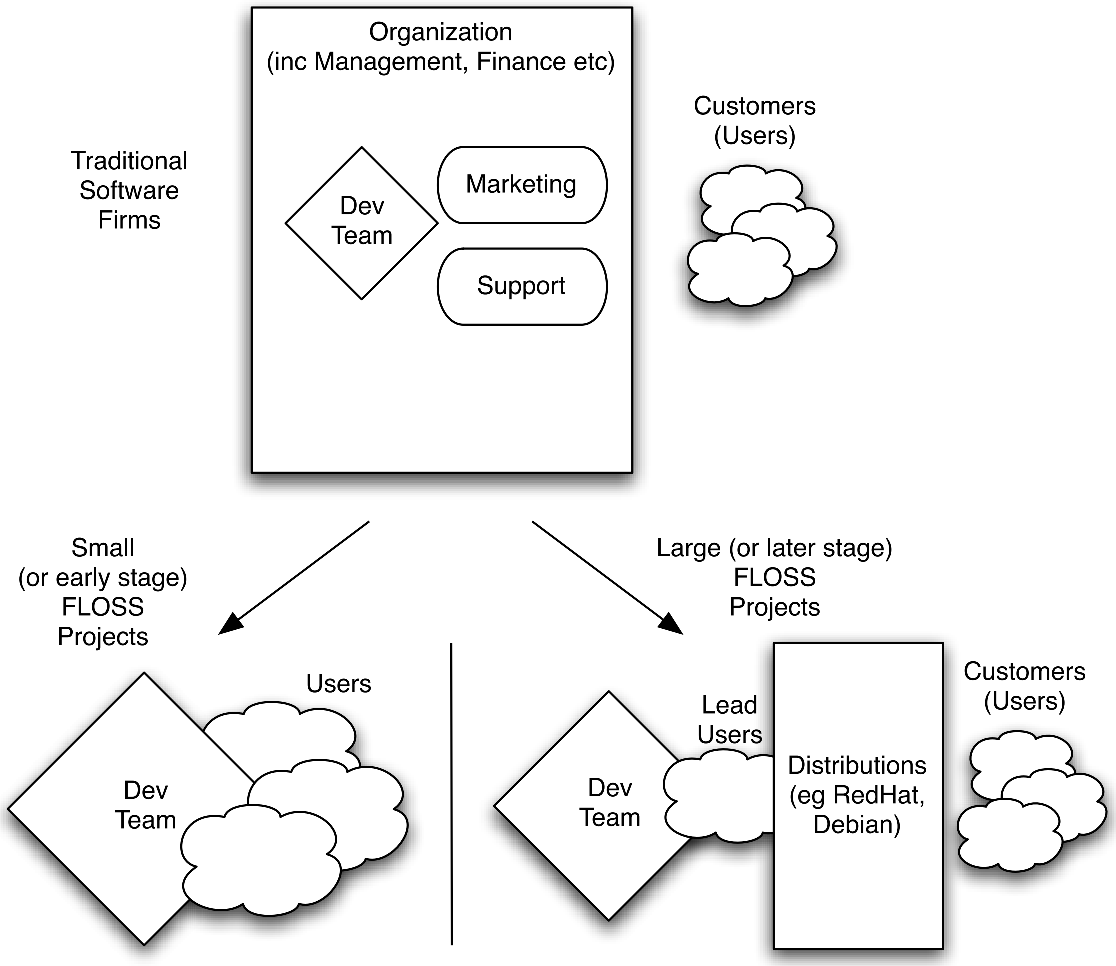
This dissertation is important and timely because it asks and answers important questions about an important, surprising and interesting phenomenon. If we are to understand FLOSS development well enough to envision the future of work and perhaps adapt these understandings to the problem of collective action it is vital to achieve an answer which integrates two aspects: the motivation of participants and the organization of production.

The software produced by FLOSS projects is a key infra-structural component of the information systems of organizations, as well as the entire internet where the Apache webserver and the BIND domain name software are the most used among their respective competitors. At its best, FLOSS is reliable and standards compliant and it is certainly widely available and relatively cheap. For the same reasons it is useful it is also a threat to the business models of proprietary software companies, in as much as they rely on selling licenses and their near monopoly of integration and support services for their products. FLOSS is also having global economic effects because it allows enterprises in developing countries to obtain solid infrastructure inexpensively. This frees their limited resources to concentrate on competing to produce higher value-added products and services.

The artifacts these projects are creating are impressive but equally impressive, from a research point of view, is that they are able to succeed in circumstances that the management literature has found to be challenging. These circumstances include working at a distance over computer media, working without formal leadership and drawing together participants, including volunteers, with diverse motivations and adapting to the changing mix of available participants, time and context (Lipnack and Stamps, 1997; Powell et al., 2004; Olson et al., 2002; Scheier, 1977). Somehow these projects are able to find enough order in this potential chaos; to invent or adapt ways of working that keep the project successful.

Of course any software package is limited in usefulness without support. With proprietary software that means interacting with the vendor, their support partners and, perhaps, a user-community (such as Microsoft's developer network). This interface is already complex enough to have interested Information Systems researchers for many decades. The introduction of the FLOSS context adds significant complexity. Figure 1.1 attempts to sketch this, showing the changing boundaries and the increasingly close relationship between users and the development team. FLOSS communities are not like traditional software venders and are sometimes defensive and ornery (Crowston and Howison, 2006). When, however, they are understood they can be very innovative and supportive. Adding to the complexity, software development companies that have employees with strong FLOSS experience can find those employees to be highly mobile, backed by their standing as individuals in the FLOSS community. Therefore understanding how to approach and to work with a

Figure 1.1: Shifts in the macro environment of software production


Organization
(inc Management, Finance etc)
Traditional Software Firms
Dev Team
Marketing
Support
Customers (Users)
Small (or early stage) FLOSS Projects
Users
Dev Team
Large (or later stage) FLOSS Projects
Lead Users
Dev Team
Distributions (eg RedHat, Debian)
Customers (Users)

FLOSS community is increasingly a strategic imperative for any organization which develops or supports software (Crowston and Howison, 2006).

FLOSS and its development also speaks to a long-standing theoretical question in Organization Science: the origin of interdependencies in work. These have long been held as a primary determinant of the appropriate coordination mechanism and thus organizational structure (e.g. March and Simon, 1958; Thompson, 1967; Mintzberg, 1979; Malone and Crowston, 1994). Recent work in longitudinal studies of naturally occurring work has questioned the "purely structural" nature of task interdependency, arguing that interdependency is an emergent property of collaboration (e.g. Wageman, 1995; Wageman and Gordon, 2005; Faraj and Xiao, 2006). The work in this dissertation speaks directly to this issue, adding a focus on the resource environment and motivations of participants as a determinant of interdependency.

1.3 Scope

Up to this point we have discussed FLOSS and its development as though it were a single phenomenon, a common but mistaken position (Nakakoji and Yamamoto, 2001; Crowston and Howison, 2006; Krishnamurthy, 2002). The label itself refers only to the licenses under which software code is released, not to the manner in which the projects obtain their resources or build software. Falling under the single label of FLOSS are projects as diverse as a five line script written and released by a single author to publicly listed companies with thousands of employees, such as RedHat. The license is not unimportant, indeed in the Discussion (Chapter 8) its stipulations are an important condition for the model of organizing described in this dissertation, yet the license is not sufficient to bound a phenomenon for useful research.

The understanding of FLOSS development is complicated by the many hybrids it forms with other types of organization. For example some important projects, like the Firefox browser and OpenOffice, began life as for-profit, co-located software development and retain some of that structure in their organization. Many projects have substantial numbers of employees who are paid, directly or in-directly by businesses for their work on the project. Such employees are often physically co-located. The impact of such aspects is weakly understood, yet it is clear that they are hybrids between relatively well-studied forms of organization, like traditional for-profit production firms, and “something else”, something novel.

The research, findings and theory outlined in this dissertation focus on that “something else”. The cases chosen throughout the dissertation are cases of “community-based" FLOSS development. By this we mean that they are projects that began "in the community"; they are not associated with any other type of institution, such as a firm or a foundation. Such projects are all exclusively volunteer: no-one is paid directly for their involvement in the project. And they are entirely distributed projects: there is no geographical centre, such as an office, where participants regularly meet face to face. None of the projects handle revenue from selling their software or services (although individual participants might be paid by organizations for their skills or experience from time to time). Each of the projects adopts more-or-less the same technical infrastructure: a publicly-readable source code repository, mailing lists or forums, trackers and regular releases documented with release notes. In this way they are similar to the origins of Linux, *the* critical case of FLOSS development.

One objection to this choice is that the cases chosen are small and relatively unimportant as pieces of software. There is truth in this: none are powering the internet or large, expensive websites. Yet the projects studied in this dissertation are relatively large, at least in terms of numbers of developers, and are popular given the projects' limited aims. Each challenges multiple commercial applications. Most importantly their scale and non-hybridization provides the right empirical environment for untangling the novelty that is central to what is meant when FLOSS is invoked in discussions of organizing. Understanding the organization of community-based projects is vital to understanding adaptation, including hybridization.

Diverse fields, both academic and practitioner, look to FLOSS for potentially radical improvements even if their understandings are perhaps based more on perceptions than reality. Whether the surprise be in FLOSS intellectual property policies, or the common lack of formal over-arching organizations, or the heavy involvement of unpaid volunteers working in their spare time, or the informality of the work arrangements, or the openness of project decision making, or the innovative use of collaboration technologies, the message is the same: we can learn from how these teams and communities work. It is, therefore, imperative for research to investigate, question and test these common perceptions.

It is hoped that this research will be useful to both academic and practitioner audiences. The intended academic audience includes researchers from Information Systems, Organizational Science, as well as Small Group researchers from Social Psychology and empirical Software Engineering researchers. Specifically:

1. Information Systems scholars interested in the organization of technology-supported teams, as well as those interested in understanding the strategic and operational implications of FLOSS for the IS function and software development firms,
2. Organizational scholars interested in novel organizational forms and sources of interdependency as a way to understand appropriate organization.
3. Software Engineering researchers interested in the interaction of organizing and the task of building software.

The practitioner audiences for this research includes those with and without experience in FLOSS projects, specifically:

1. IS professionals seeking to engage effectively with FLOSS communities,
2. Software developers seeking to start or effectively contribute to a FLOSS project,
3. Experienced FLOSS participants seeking to adjust their project to meet the changing challenges as their project grows and develops.

Answers to the research questions of this dissertation are vital to a satisfactory and adaptable understanding of FLOSS development. The core of organizing is bringing together people (motivation) to build something together (production). As Literature Review I (Chapter 2) will argue, existing studies of FLOSS have failed to adequately integrate these aspects of organizing. Furthermore it is clear that technologies of collaboration and production are important, a point taken up in detail in the Discussion (Chapter 8). Only by bringing these aspects together will a more satisfying theory of FLOSS organizing emerge.

1.4 Method and ethics in the FLOSS context

The community-based FLOSS projects that are the focus of this dissertation are voluntary and transparent communities. The participants invest their time in the project for whatever purposes might drive that participant and everybody is careful to acknowledge each other's "real lives". The ethics of the community, as depicted in practitioner written guides such as Raymond and Moen (2001, §Before You Ask) are clear in arguing that people seeking input from the community should have "done their homework" with the available resources and demonstrated that they are willing to invest their own time before drawing on the time of others.

The research interest in FLOSS has been intense and projects—always pressed for time—are quick to lose patience with researchers who do not acknowledge the ethics of the community and who demand the time and attention of the participants without contributing or having adequately examined the public archives before beginning research. Requests for surveys and interviews are often greeted with annoyance, even when introduced by community leaders, and ignored as a matter of course by key participants. Two threads from the Gaim project illustrate the issue.

> *[An] endless stream of "surveys" about open source development from researchers who obviously aren't checking the background material, leading them to ask the same questions over and over to hundreds of open source developers who have better things to do with their time (e.g. developing open source applications) than Yet Another Survey with the Same Questions On It.* [3]

One participant facetiously adds a question to an imagined survey,

> *Question 20: When you became a contributor to Open Source software, did you anticipate being asked to participate in an almost identical time-consuming survey every month or so, by unconnected people from universities across the world, apparently condemned to attempt repeating the*

[3]http://sourceforge.net/mailarchive/message.php?msg_id=20050405171258.130.qmail%40web50601.mail.yahoo.com

same piece of research again and again and again, only to find that the responses are underwhelming and probably not representative? [4]

For these reasons this dissertation pursued only non-intrusive methods which do not subtract from the time available for the project and are in-keeping with the ethics of the projects. These methods are participant observation and study of fully public archives. This does mean that the research has not benefitted directly from feedback from participants; this is a loss. To counter this, and to reciprocate the projects' openness, this dissertation, its data and all intermediate products and code are being made publicly available.[5] In this way the author hopes to attract the interest and feedback of the projects studied and improve the research presented here.

1.5 The structure of this Dissertation

The dissertation is structured in three Parts. Part I contains this introductory chapter and an initial literature review. Part II is the main substance of the dissertation. It proceeds through a logic of discovery, replication and formalization, with a return to the literature between each step giving a total of five chapters. Part III of the dissertation consists of two chapters: a Discussion, which draws together answers to each of the research questions and a Conclusion, which summarizes contributions and makes the case for future research.

The initial literature review, Chapter 2, introduces a framework for studying effective teams as a starting point for research. The chapter uses the framework to organize a high-level survey of existing literature on FLOSS development. This survey highlights a split between studies with an interest in motivation and those

[4]http://sourceforge.net/mailarchive/message.php?msg_name=2c5131b00506060210300a1849 %40mail.gmail.com#content

[5]http://james.howison.name/pubs/dissertation.html

with an interest in production and argues that this is why the literature has not yet been able to provide satisfying answers to the FLOSS phenomenon.

Part II presents an unfolding arc of discovery, replication and formalization, with returns to the literature as necessary. Each study describes its own method, results and interim conclusions. Discovery: BibDesk Participant Observation (Chapter 3) provides discovery through participant observation, highlighting emic findings regarding the organization of FLOSS work. Such rich case study results are strengthened if they can be replicated in a systematic way (Yin, 1994). To prepare for such a replication Chapter 4 returns to the literature, updating the teamwork framework with an increased awareness of episodic work and introducing literature on coordination and dependency as an appropriate way to systematically represent FLOSS organization. Replication: Fire and Gaim Archive Study (Chapter 5) then presents a systematic archival study of two projects that are similar to BibDesk. This study replicates, focuses and strengthens two of the findings of Discovery: BibDesk Participant Observation (Chapter 3): the dominance of individual work and deferral as a production strategy.

At this point the dissertation has a tentative answer to the first research question: effective FLOSS production is primarily organized into short, individual tasks which advance the software through layering. Yet the question remains: how is FLOSS production able to integrate and advance through such work and how is this linked to motivations? Chapter 6 returns once more to the literature to examine motivation and the literature that links it with production. This background enables Chapter 7 to present a more formalized model which "generalizes to theory" (Yin, 1994), explaining the findings of Chapter 3 and Chapter 5 and integrating understandings of production and motivation in FLOSS projects, thereby answering RQ2. The model presented makes explicit the conditions necessary for it to operate and thus provides a framework to answer to RQ3, on adaptation, which is undertaken in the Discussion

(Chapter 8). In this way the dissertation answers all of its research questions and advances our understanding of FLOSS development and its implications for the future of work.

Chapter 2

Literature Review I

In order to study the effective organization of successful FLOSS production, and link it to motivation, it is useful to adopt a framework for understanding existing literature. The framework chosen is the Input-Process-Output (IPO) framework and its descendants. This framework identifies attributes of the empirical world which form part of an explanation of effective teamwork. While the "teamness" of FLOSS production is a central question of this dissertation, one has to start somewhere and the literature on teams, rather than organizations, individuals or societies, is the approach found to be most relevant, in that it assisted the author most in understanding and framing his participant observation experiences. The framework comes from the field of Management and is heavily influenced by social-psychology. It is quite positivist in nature. This fits with the majority of the literature on FLOSS and is useful for this reason, even though it would not cope well with the FLOSS literature from more interpretative and critical fields, such as Science and Technology Studies, which are not examined here.

The IPO framework introduced in this chapter allows us to present a high-level summary of the FLOSS literature, drawing heavily on an as yet unpublished review article to which the author contributed (Crowston et al., 2008). However, the presentation here goes further in order to addresses the research questions of this dissertation. It argues that current research on FLOSS does not adequately link understandings of motivation and the organization of production, and therefore fails to provide a substantive basis for a satisfactory explanation of the FLOSS phenomenon.

2.1 A framework for studying team effectiveness

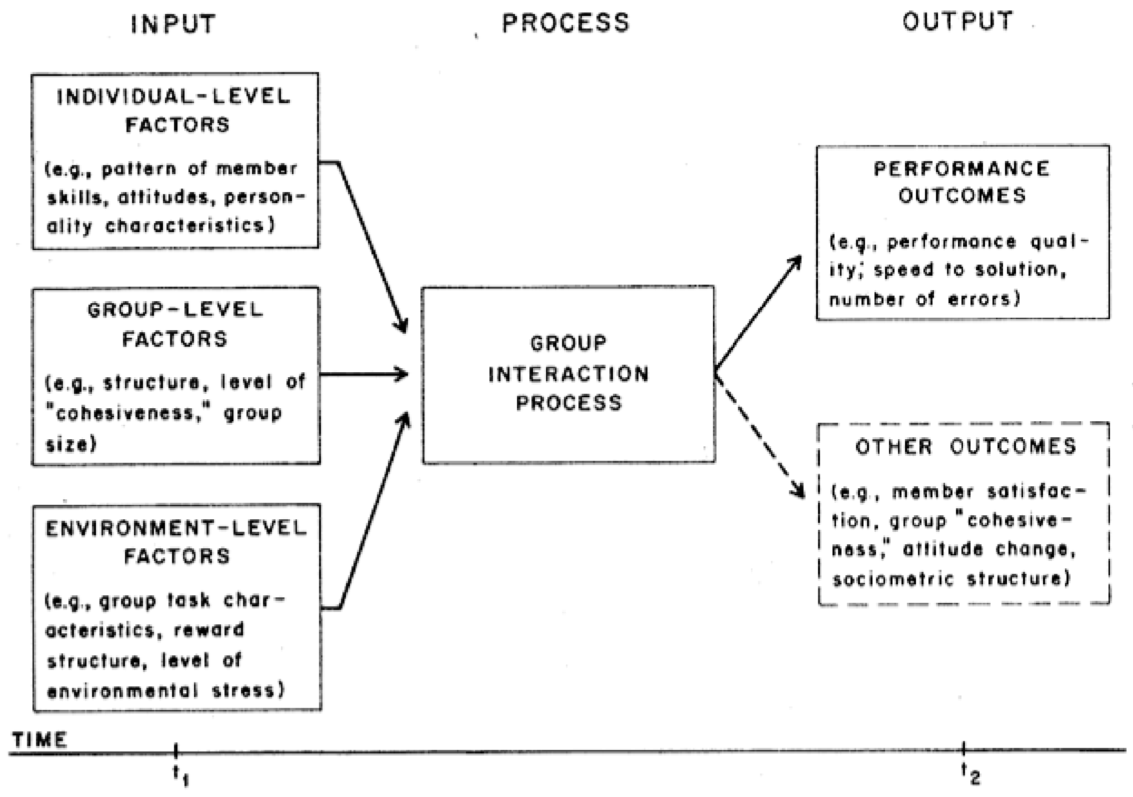
The Input-Process-Output framework (IPO)

A crucial approach to understanding the organization of teams is part of a broader approach to explaining what makes for effective teamwork. The Input-Process-Output model of team effectiveness was introduced and developed by Richard Hackman and colleagues (Hackman and Morris, 1978) as well as Joseph McGrath (McGrath, 1991, 1984). As shown in Figure 2.1 work teams are conceived of as converting inputs to performance outcomes, and a team's processes are what mediates that conversion. Inputs include the team's composition, structure and task (or goals) as well as the environment in which they operate. Outputs include the results of a team's interactions, including task output (e.g. software) as well as effects on the team itself (such as satisfaction and intention to continue to work together).

Focusing on Process

The Process category has been significantly fleshed out since this early representation and Rousseau et al. (2006) provides a comprehensive review. It begins by defining them as bringing "together all of the behavioral, cognitive, and affective phenomena existing in teams" before focusing on behaviors, as they are "observable and measurable actions" (p. 541).

Figure 2.1: An early version of Hackman's Input-Process-Output Framework (Hackman and Morris, 1978)


INPUT
PROCESS
OUTPUT
INDIVIDUAL-LEVEL FACTORS
(e.g., pattern of member skills, attitudes, personality characteristics)
GROUP-LEVEL FACTORS
(e.g., structure, level of "cohesiveness," group size)
ENVIRONMENT-LEVEL FACTORS
(e.g., group task characteristics, reward structure, level of environmental stress)
GROUP INTERACTION PROCESS
PERFORMANCE OUTCOMES
(e.g., performance quality; speed to solution, number of errors)
OTHER OUTCOMES
(e.g., member satisfaction, group "cohesiveness," attitude change, sociometric structure)
TIME
t_1
t_2

These behaviors are divided up into "taskwork" behaviors and "teamwork" behaviors. Taskwork behaviors "contribute directly to the accomplishment of the tasks and are related to the technical aspects of the tasks that exist independently of work organization" which, in Rousseau's opinion, "may not be generalized to other [types of tasks that other teams perform]" (p. 542).

By contrast teamwork behaviors "are inherent to the existence of work teams" and are "the overt actions and verbal statements displayed during interactions between team members to ensure a successful collective action" (p. 542). Teamwork are those actions that make the team work *as a team.* The anticipated generalizability of teamwork, as opposed to taskwork, has reinforced the focus of this literature on teamwork, rather than taskwork, a question to which we will return below.

Figure 2.2: A detailed conceptualization of teamwork in the IPO approach (from Rousseau et al., 2006). Note that Taskwork is set aside prior to this breakout.

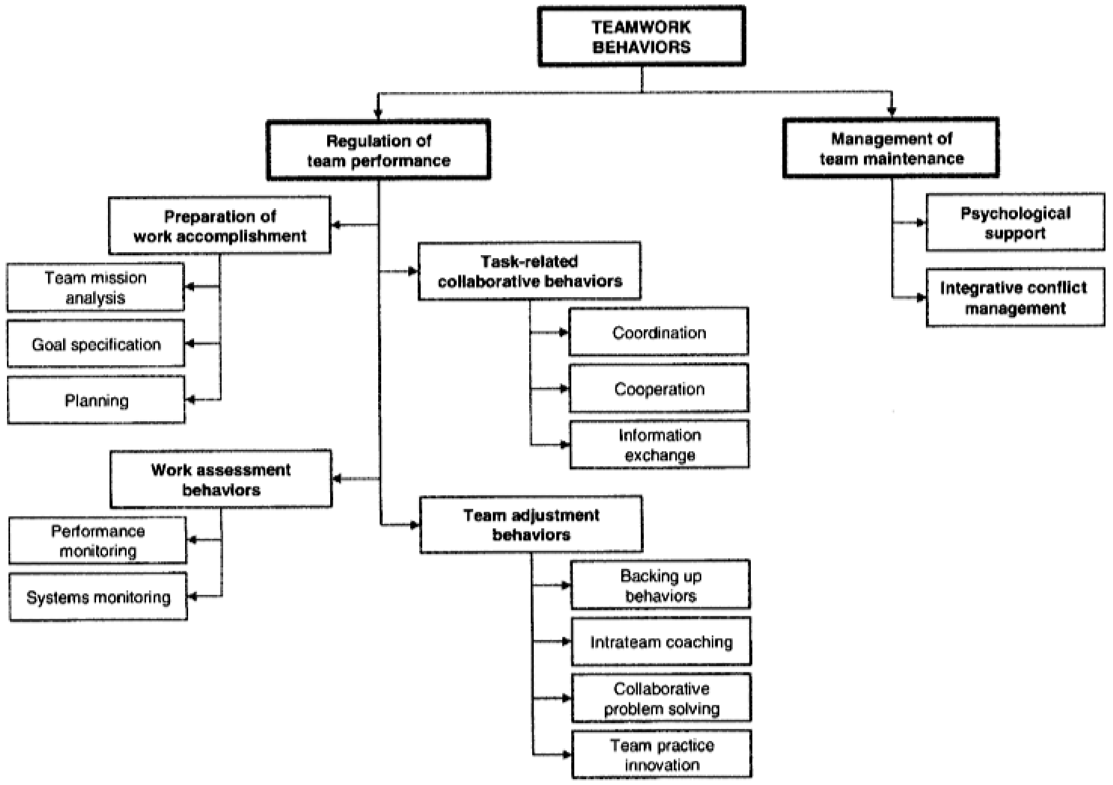

TEAMWORK BEHAVIORS
Regulation of team performance
Management of team maintenance
Preparation of work accomplishment
Team mission analysis
Goal specification
Planning
Work assessment behaviors
Performance monitoring
Systems monitoring
Task-related collaborative behaviors
Coordination
Cooperation
Information exchange
Team adjustment behaviors
Backing up behaviors
Intrateam coaching
Collaborative problem solving
Team practice innovation
Psychological support
Integrative conflict management

Figure 2.2 shows the summary of teamwork presented by Rousseau et al. (2006) as part of the IPO approach to Process. Their tabular summary of processes and associated authors and studies runs to two pages and includes such diverse areas of study as communication, cooperation, helping behavior, innovating, conflict management, goal setting, monitoring, problem solving, adapting and decision making (to name just a few). Communication is considered part of the "information exchange" category.

The research in this dissertation requires a way to understand the organization of production and link it to motivation so that its adaptability can be assessed. The teamwork processes in the IPO model are clearly linked to organization, but it is also clear that they would need to be represented and understood in a variety of different manners. Scholars do not, in general, attempt to represent and examine each of these concepts, instead focusing on a small number in great detail and developing a representation of that specific process. For example, Crowston and Osborn (2003) describes techniques to represent coordination practices, and Poole and Roth (1989) developed a representation for decision making practices.

Despite the depiction of these aspects of work in teams as “observable behaviors” it seems that a substantial, and specialized, conceptual apparatus is required to turn observations of human actions into knowledge about these aspects of how teams work. The effort to see these aspects as ‘of a kind’ requires a move to a relatively high conceptual level and the path to operationalization and observation is certainly different for each. Researchers need to look in different places and in different ways to see each of these processes.

The setting aside of taskwork is particularly telling in this regard. Certainly some of these processes occur through explicit actions oriented to the team-as-team but it seems likely that many of these functions are, in fact, achieved through and during (or at least at the same time and in the same venues as) taskwork.

2.2 Current FLOSS research

The research on FLOSS development is substantial. This section presents a summary of the literature drawn from Crowston et al. (2008), a review article to which the author contributed. This review is presented in the context of the IPO model presented above, beginning with Inputs then discussing Outputs before dealing with Process in greater depth. The section concludes by going beyond the review article to argue that there has been little to no integration across these categories, specifically there has been little to no integration between studies of motivation and production.

Inputs (FLOSS Context)

Crowston et al. (2008) discuss three main categories of Inputs: member characteristics, project characteristics and technologies in use. The research on member characteristics includes work on the motivations and time commitments of individual participants as well as developer numbers.

The empirical literature on FLOSS motivations is extensive and growing (Lakhani and von Hippel, 2003; Lakhani and Wolf, 2003; Hars and Ou, 2002; Ghosh et al., 2002; Ye and Kishida, 2003; Chin and Cooke, 2004; Ke and Zhang, 2008). One clear finding is that the motivations of participants are diverse and there is little uniformity amongst participants (Feller et al., 2005). The diverse motivations found in surveys of FLOSS participants fall into two overall categories. The first are motivations that could be obtained in an individual activity context:

- learning by doing
- the product itself
- intellectual stimulation
- self-efficacy

The second are those where motivations require associational activity and likely be better supported by interdependent performance.

- learning from others
- building reputation
- norm-based reciprocity

In addition there are motivations that don't fall into either category, including ideological commitment to opposing proprietary software.

Hertel (2007) points out that these studies of motivations in FLOSS projects have been almost exclusively focused on "person-oriented" rather than "job-related" factors, by which he means aspects of the experience of participation, such as autonomy, task identity, and feedback opportunities (Hackman and Oldham, 1980). He argues job design factors are more relevant to adapting FLOSS organization techniques to other environments, since they are more amenable to manipulation. There is, however, little empirical work examining these factors. Hertel (2007) is a conceptual paper and is "not aware of any systematic job design analyses of OSS being available to date". Chin and Cooke (2004) do include job design and coordination in their model, but the only data presented is a pilot study of under 40 respondents selected at a face to face open source support group. Few of their hypotheses were supported. Ke and Zhang (2008), a very recent contribution, used a survey to find that task empowerment (assessments of autonomy and meaningfulness) was positively related to task effort.

The stronger research in this area includes a measure of effort expended by participants, allowing the research to assess which motivations were more "active". Research without such a dependent variable tends to result in something of a 'laundry list' of motivations and relies on frequency of mention to rank order them. Lakhani and Wolf (2003) use their measure of effort to emphasize the importance of two motivations in particular: "enjoyment based intrinsic motivation" and the "need for the product". This finding was echoed by a content analytic study of open-ended survey responses (Hemetsberger, 2001). Not all the studies are in agreement however. Hertel et al. (2003), in a study of the Linux community, found greater emphasis on community identification particularly with groups building in sub-systems of that very large project. It is worth noting, as Crowston and Fagnot (2008) do, that none of these studies attempted to separate initial motivations (recruitment motivations) from motivations for on-going participation (retention motivations) despite the possibility that these might be distinct.

There are few studies that have examined overall time spent on FLOSS projects. Luthiger (2004), using self-reported individual measures of time commitment, found an average of 12.15 hours per week spent. Project leaders were found to spend an average of 14.13 hours, and bug-fixers and otherwise active users spending closer to 5 hours per week. These findings are complimented by those of Lakhani and von Hippel (2003) who studied the amount of time participants spend reading and answering user support questions in the Apache forums, finding that the most frequent answer providers spent 1.5 hours per week, dropping to just half an hour for infrequent information seekers. This confirms the common understanding that FLOSS development happens in "spare" time and, in general, is not the central undertaking in participant's lives.

Another crucial input in the IPO framework are the technologies of production and task. Many studies of FLOSS development point to the importance of collaboration technologies, including source code repositories and email lists (Fogel, 1999; Scacchi, 2004) as well as Trackers (Michlmayr, 2004). Robbins (2002) examines nine FLOSS technologies and argues that they "fit the characteristics of open source development processes" and that their adoption in other software environments may influence processes in those environments. Yamauchi et al. (2000) argue that the patterns of production they observe (discussed under Process below) are the result of "collaboration with lean media", drawing on theories of media richness.

Process

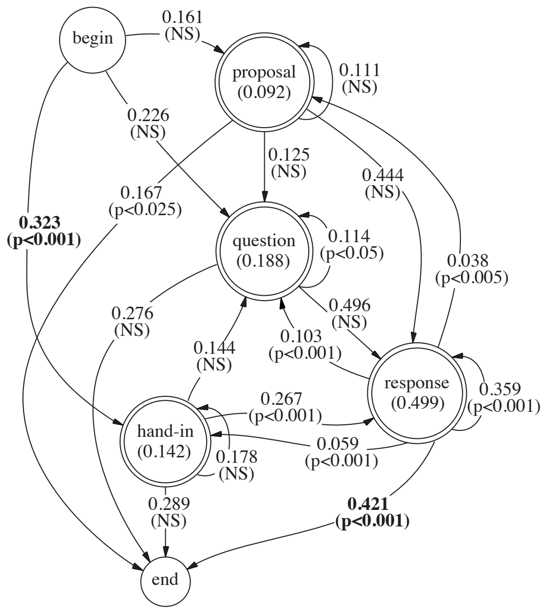
The work examining process can be usefully summarized using the division between teamwork and taskwork described above (Rousseau et al., 2006). Studies of teamwork have focused on separated aspects such as Coordination (Crowston et al., 2005), Learning (Annabi, 2005), Socialization (von Krogh et al., 2003), Decision-making (Heckman et al., 2006; Gacek and Arief, 2004; Fielding, 1999). These studies are useful but rarely discuss how their chosen process of interest interacts with the rest of the context of FLOSS development.

Taskwork, in the FLOSS context, refers to the day to day production of software and has been a less common topic of study. One group of research has worked to compare FLOSS projects to established models of software production, such as traditional waterfall production and agile methods (Jensen and Scacchi, 2007; Mockus et al., 2002; Warstaa and Abrahamsson, 2003).

Only a few studies have actually examined the unfolding course of work. The first is Yamauchi et al. (2000) who conducted content analysis of the mailing lists of four projects and built a process model of the origins of new features. They employed a state-transition probability analysis, shown in Figure 2.3, to conclude that the new feature process often begins with a *code hand in*, rather than planning and resource assignment, followed by coordination to integrate the code. They argued that this process is caused by only having “lean media” available and thus reducing the ability of the project to work in other ways. They appear to assume that the project would be more successful if it worked in ways that promoted greater collaboration, but that the leanness of the available media prevents this. In short the developers make “the best of a bad situation”.

Crowston and Scozzi (2004) conducted a study which examined episodes of bug-fixing in a Bug Tracker, coding each phase of the work and summarizing the processes found in terms of sequences and dependencies between steps. They did examine participation in their bug-fixing tasks, noting that some participants seemed to do substantially more work and that most bug-fixing processes involved only one or two core members in interaction with active users in the wider community. Heckman

Figure 2.3: The transition probabilities between events on the newconfig project developer mailing list. Note the high occurrence of the transition from begin to hand-in (line down left of figure) (Yamauchi et al., 2000, Fig. 3).



et al. (2006) take a similar approach, examining the unfolding of episodes of software implementation, which they conceptualize as a type of decision-making. Nonetheless they are able to report sequential patterns and durations as well as participation figures, finding relatively low numbers of distinct participants per episode. Both of these studies examined only a single venue—either the bug tracker or the mailing list—and neither attempt to explain their findings through a theory that integrates important aspects of the full context of FLOSS production, such as participant motivations.

Outputs (FLOSS Success)

The IPO model above identifies both products and developer satisfaction as outputs of teamwork and the whole framework is designed to describe the factors contributing to effective teamwork. The question of effectiveness is thus crucial, since this dissertation is interested in successful FLOSS projects. Since success is the most common dependent variable in studies of FLOSS development it has been extensively conceptualized.

Crowston et al. (2003, 2006a) examine the question of the meaning of success and effectiveness in the FLOSS context. They examined the existing literature and conducted an online focus group as well as developing sample success measures. They argue that, in the FLOSS context, success is most usefully understood as a multi-dimensional construct extending across each of the phases of Input, Process and Output, rather than the traditional focus on Outputs. This is because FLOSS projects are not pre-assigned and externally managed tools for production, they are entities which must be self-defining and self-sustaining if they are to achieve their own goals and have substantial impact. They propose measuring success through measures of inputs (such as developer attractiveness), measures of process (such as the speed of fixing bugs), and measures of output (such as careful use of downloads and pageviews to show popularity and the quality of the software produced).

Unfortunately success is rarely conceptualized in such a complex way in the rest of the literature, with research choosing to concentrate on one measure determined by the overall concern of their academic field. For example business focused research is often content to use downloads or popularity of software as a proxy (e.g. Krishnamurthy, 2002; Stewart et al., 2006). It seems that this is, in part, because downloads are considered analogous to sales of closed-source commercial software. Similarly Software Engineering focused research is often content to use system quality, such as maintainability of the software as a measure of successful FLOSS production (e.g. Schach et al., 2003; Bezroukov, 1999).

2.3 Linking motivation and production

Crowston et al. (2003, 2006a) rest their conceptualization of success on an argument which begins to analyze the full challenge facing FLOSS projects, from motivation and recruitment of developers, to the satisfying creation and distribution of working software. The authors have not yet developed integrated empirical tests of their conceptualization and few other studies have attempted this type of integrated explanation.

The work of Michlmayr argues strongly for the importance of the volunteer context in the organization of FLOSS production (Michlmayr, 2003, 2004; Senyard and Michlmayr, 2004). Prior to pursing an academic Ph.D., Michlmayr was the elected leader of the Debian FLOSS distribution and brought this experience to his work. He argued that the volunteer context strongly shaped the organization of production,

> *There is an important discrepancy in the requirements of quality for planned and systematic activities and the voluntary nature of open source projects which is based much more on ad hoc activities" (Michlmayr, 2005).*

Senyard and Michlmayr (2004) argue for a life-cycle model of FLOSS development that began with the need to produce a working prototype to attract participants and then argue that successful projects ensure that their code is able to support satisfying individual and parallel activity which they, following traditional software engineering understandings, ascribe to modularity, "The properties of the design and implementation of the prototype must motivate others to participate in the project. This is facilitated by a simple, clear and modular design." (p. 5).

Finally Michlmayr (2003) examines difficulties within the Debian project due to over-reliance on individual volunteers as maintainers for packages. Interestingly his proposed solution is not the obvious course of "replacing lead developers with teams or expanding the size of existing teams" since groups might reduce the participants' motivations by reducing feelings of ownership and responsibility, leading to assumptions that others will do it. Further they argue against using teams because they introduce complexity and communication issues. Overall Michlmayr argues that team responsibility “works to counter the benefits which have made Free Software community’s ‘one and many’ development and debugging system so effective.”.

Rather Debian adopted the simple expedient of an “explicit backup maintainer”; another individual. In addition they provided effective, publicly visible communications so that the backup individual can assume the role if it is not being performed. Even though Michlmayr is concerned that “strong reliance on individual developers is a quality assurance consideration in that it is unrealistic to expect complete predictability and reliability from volunteers”, his solution is to acknowledge the motivational importance of the individual organization of FLOSS work over the more traditional management technique of creating groups, while stressing a communications infrastructure which facilitates opportunistic collaboration.

Other than these contributions, and three motivational studies addressing job design discussed above, the literature on motivations in FLOSS does not consider their impact on production, while the small literature on processes in production has only rarely examined theoretical reasons for the shape of the processes they observe.

2.4 Literature I: Conclusions

This literature review makes the argument that the FLOSS literature has generally failed to integrate research on motivation and organization and that this seems fundamental to understanding how effective FLOSS development projects do what they do. The framework and literature examined here show that the phenomenon is not wholly unexplored, but they also indicate the peculiar and sometimes counter-intuitive nature of FLOSS projects and show the importance of the participant’s own understanding and experience. Further, the review shows that there are a myriad of possibly relevant aspects of organization and clearly any study cannot address all of them. Finally, as discussed in the Introduction, we take an ethical position that research should be as non-intrusive as possible. For these reasons the research begins with participant observation, discovering results which will then be replicated through archival study before being formalized into an explanatory model. Together these approaches are capable of answering the research questions of this dissertation and improving our understanding of FLOSS development and its organizational implications.

Part II

Research

Chapter 3

Discovery: BibDesk Participant Observation

3.1 Introduction

This chapter reports the results of participant observation of a FLOSS project called BibDesk, over a period of four years. The presentation of results is organized into four sections. The first section presents the case study method: sensitizing concepts, case selection and participant-observation logistics. The next two sections provide a rich description of the case from two different perspectives, that of an observer and that of a participant. In the fourth and final section the chapter sums up findings for each of the sensitizing concepts.

This chapter is written in the first person to emphasize that it is the result of individual participation and observation. The experience that guides the results of this chapter is personal and specific, thus the first person "I" is appropriate and reminds the reader of the nature of the research.

3.2 Method

Sensitizing Concepts

Case studies, and particularly those with a participation component, provide a wealth of immerse, holistic experience. The challenge for the researcher is to embrace this holism, without loosing sight of the ultimate goal of producing structured, transferable knowledge. For this reason I chose to enter the field with three concepts through which I wished to focus my experience in the field. Such priming of the interpretative process is recommended for qualitative research (Glaser, 1978; Patton, 2002; Bowen, 2006).

The sensitizing concepts are linked to the overall research questions of the dissertation and to the literature review on FLOSS. They are: 1) motivation, 2) coordination and dependency, and 3) the experience of organization. During the case study I periodically returned to the literature, particularly that presented in the proceeding and following chapters, so as I entered the field I had less comprehensive understanding of the concepts. Indeed my experience in BibDesk helped guide my integration of the literature, just as the literature guided the results presented in this chapter.

Selecting a case

A case study needs a case; and that case has to be a case of something. In the Introduction to this dissertation it was argued that the most appropriate type of FLOSS project to discover the novel "something else" at play is a community-based, non-hybrid project. Further I needed to choose a case that I could realistically contribute to and which I was likely to be able to sustain a long-term commitment. For this reason I let my case emerge naturally from my day to day practice as an academic. I began to use FLOSS software for my academic work whereever possible, focusing particularly on writing and statistical analysis. For writing I switched from using

Microsoft Word to trying both OpenOffice and the LaTeX document preparation system (common in computer science), and from Endnote to BibDesk, a reference management program for Mac OS X. At the same time, while taking introductory statistics, I used the R statistics package, rather than SPSS. Further I taught myself Perl, an open source and widespread scripting language and Ruby on Rails, another scripting language and web application framework.

Over time I came to see BibDesk as an appropriate project for participant observation. I found entry to the project easier and found that it integrated most consistently into my daily practice. By contrast OpenOffice was quite opaque, as many functions were still performed by the co-located Sun engineers; it is a hybrid project. While I remain a frequent user of R, Perl and LaTeX, these projects did not prove to be appropriate for participant observation. LaTeX is a very mature project with very little continuing development. For R and Perl the substance of the day to day work, statistics and language theory, was beyond my programming skills.

By contrast BibDesk was useful to me in a daily fashion, small and emergent enough to hope to come to understand in detail and the substance of its programming task was within my understanding. Nonetheless there was, as I will show below, a significant and sometimes insurmountable learning curve as I worked to provide useful contributions to the project. As discussed below the project has always been open source, and the participants are unpaid and distributed. For these reasons BibDesk is an appropriate case of community-based FLOSS development, as defined in the Introduction (Part I, on page 8).

Study Logistics

The conduct of the study was simple, at least in terms of its logistics. The project, as with most FLOSS projects, is accessible to anyone with a computer and internet access. The technical infrastructure, as we shall see, was open and available; although my experience in learning its use was crucial to understanding how the project works and is reported below.

During the course of the participant observation I recorded my growing understandings in two ways. The first is to note that the nature of the collaboration means that evidence of my participant was automatically stored in the project's public archives. When this was not true, as we shall see, I recorded my efforts in a journal. In this same journal I noted episodes of activity which I thought relevant to the sensitizing concepts, together with references to the literature. Simultaneously I was reviewing the relevant literature, so evidence of my growing understanding of the relevant literature is also recorded in this combined journal. Finally the project archives formed a source for the pilot archival study which pre-figured the archival studies to be reported in Chapter 5.

Overall I have participated in the project for over 5 years, from May 2003 until the present (December 2008). I was quite active for about 2 years, from mid-2003 through mid-2005 and less active from then onwards. Today I continue to have commit rights but my involvement is mainly as an active user, answering user questions and assisting the active developers in discussions with other users. My participation is discussed in detail below.

I did not explicitly inform the BibDesk participants that I was conducting research, nor did I seek their informed consent. Neither did I artificially hide my area of study: it is plain on my website for all to see and I often included FLOSS research references as test cases in discussions in the email. This is ethical because FLOSS projects are entirely public and all participants know this, indeed they celebrate it. I judged the risk to other participants to be minimal and the impact of seeking informed consent substantial. There is a useful debate (Robles, 2007) regarding whether transformations of public records in certain ways, such as overall SNA or contribution league tables, provide risks to the participants or the projects, but this work does not do such overall summaries. Furthermore this dissertation and all intermediate archives and ontologies will be released in reciprocation for the project's openness. I also make available all my FLOSS publications as downloadable PDFs, ensuring that my work is available to the subjects of this research.

3.3 BibDesk as a Project

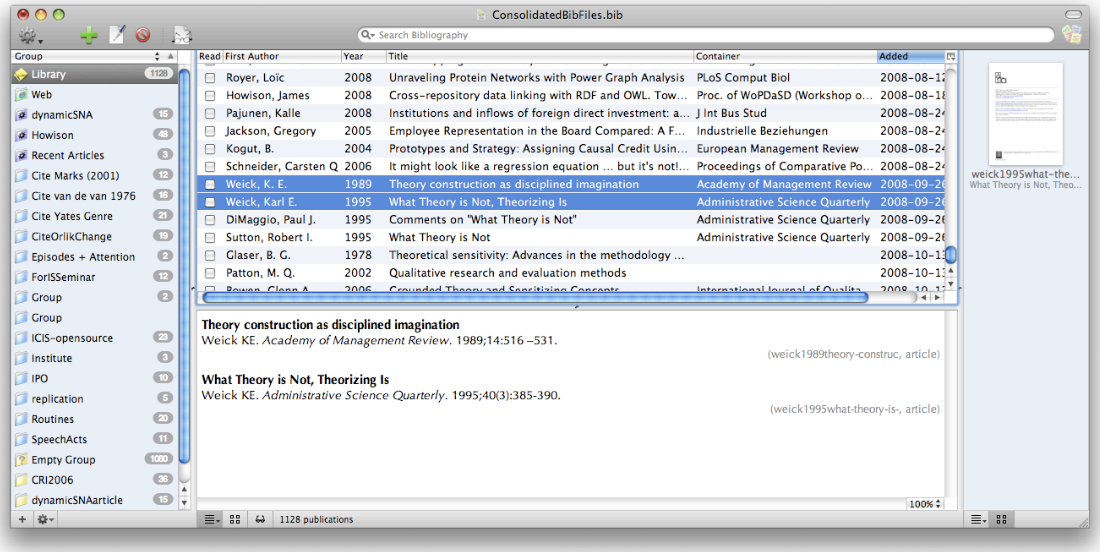
BibDesk is a bibliographic manager that runs on Apple's OS X operating system. It began as an editor for BibTeX, the primary bibliographic system used in the Natural and Computer Sciences but has become an increasingly comprehensive research management tool, an "iTunes for Academic References"; although it is bound to BibTeX in fundamental ways. Its primary proprietary competitor is Endnote, although BibDesk does not have comparable integration with Microsoft Word. There are other FLOSS reference managers, including JabRef and Zotero, which has made a large impact in the last year. However in its niche—Mac OS X users using BibTeX and LaTeX—BibDesk is a dominant choice.

The project is hosted on Sourceforge, the primary hosting provider for FLOSS projects. Sourceforge's services are available without charge to projects released under an open source license. BibDesk was created by a computer science doctoral student at UCSD and first released on Sourceforge in September 2002, as an already working, but limited and somewhat rough, BibTeX manager.

Project Success

The BibDesk project is small compared to some very well known FLOSS projects but it is relatively successful compared to the bulk of registered FLOSS projects. As discussed in Chapter 2, Crowston et al. (2003, 2006a) provide a definition of success

Figure 3.1: An October 2008 screen shot showing the basic layout of BibDesk. Note the groups on the left-hand side, the Container column, and the PDF linked to the entry on the right



Group	
Library	1128
Web	
dynamicSNA	15
Howison	48
Recent Articles	3
Cite Marks (2001)	12
Cite van de van 1976	16
Cite Yates Genre	21
CiteOrlikChange	19
Episodes + Attention	2
ForISSeminar	12
Group	2
Group	
ICIS-opensource	23
Institute	3
IPO	10
replication	5
Routines	20
SpeechActs	11
Empty Group	1080
CRI2006	36
dynamicSNAarticle	15

Read	First Author	Year	Title	Container	Added
☐	Royer, Loïc	2008	Unraveling Protein Networks with Power Graph Analysis	PLoS Comput Biol	2008-08-12
☐	Howison, James	2008	Cross-repository data linking with RDF and OWL. Tow…	Proc. of WoPDaSD (Workshop o…	2008-08-18
☐	Pajunen, Kalle	2008	Institutions and inflows of foreign direct investment: a…	J Int Bus Stud	2008-08-24
☐	Jackson, Gregory	2005	Employee Representation in the Board Compared: A F…	Industrielle Beziehungen	2008-08-24
☐	Kogut, B.	2004	Prototypes and Strategy: Assigning Causal Credit Usin…	European Management Review	2008-08-24
☐	Schneider, Carsten Q	2006	It might look like a regression equation … but it's not!…	Proceedings of Comparative Po…	2008-08-24
☐	Weick, K. E.	1989	Theory construction as disciplined imagination	Academy of Management Review	2008-09-26
☐	Weick, Karl E.	1995	What Theory is Not, Theorizing Is	Administrative Science Quarterly	2008-09-26
☐	DiMaggio, Paul J.	1995	Comments on "What Theory is Not"	Administrative Science Quarterly	2008-09-26
☐	Sutton, Robert I.	1995	What Theory is Not	Administrative Science Quarterly	2008-09-26
☐	Glaser, B. G.	1978	Theoretical sensitivity: Advances in the methodology …		2008-10-13
☐	Patton, M. Q.	2002	Qualitative research and evaluation methods		2008-10-13
☐	Bowen, Glenn A.	2006	Grounded Theory and Sensitizing Concepts	International Journal of Qualita	2008-10-1

Theory construction as disciplined imagination
Weick KE. *Academy of Management Review*. 1989;14:516 –531.
(weick1989theory-construc, article)

What Theory is Not, Theorizing Is
Weick KE. *Administrative Science Quarterly*. 1995;40(3):385-390.
(weick1995what-theory-is-, article)

for FLOSS projects which takes into account measures of recruitment, process and use.

In terms of recruitment, the development team has grown organically and Sourceforge, as of October 2008, lists 13 developers. This places it in the very top percentile of projects hosted at Sourceforge. Interestingly, however, although 13 developers are listed, realistically the project currently has only three core developers and two active peripheral developers and has rarely risen above such numbers. The peripheral participants have rotated through over time but are not removed, yielding the 13 developer number. Three developers have Administrator status, meaning they can add/remove other developers and alter the tools used by the project. These three are the original founder (who has been less active for the past three years) and the two most highly active developers. There is no evidence that any of the developers are co-located and or that any have met met face to face. Indeed when I was visiting the city where the founder was a PhD student I suggested meeting socially but he declined.

In terms of processes, the project has always been amongst the top 3% activity percentile of Sourceforge projects and, as of October 2008, is in the 99.73 activity percentile. The activity percentile is a measurement of development activity, based on source code and project communications.

In terms of use, the project has consistently generated about 200 downloads a day, and has done so since early 2005. Perhaps more importantly it is well-known and used in its relatively small niche. Despite many requests, the project has never moved beyond the Mac OS X platform. Mac OS X is a relatively niche platform, but it is well represented amongst students, academics and computer scientists, who are the primary users of reference programs and LaTeX/BibTeX in particular. BibDesk has also resulted in the spin-off of a complementary, but separate, project for reading and annotating PDFs, called Skim (skim-app on Sourceforge). Like BibDesk this project began as a private project by the original BibDesk developer which was taken open in the BibDesk community. Skim is now almost exclusively maintained by one of the two other core BibDesk developers, although I believe that all of the long-term participants in BibDesk are also participants in the Skim community.

Project Infrastructure

The BibDesk project makes use of a set of technical and communication tools, drawing heavily on those offered by Sourceforge but complementing those with its own hosted resources. Sourceforge provides hosting for application releases, source code repositories, mailing lists and Trackers (structured web forms for users and developers to track bug reports, requests for enhancement and other longer term issues). BibDesk has two interactive mailing lists, the “develop” list and the “users” list and two automated lists, “commits” which broadcasts changes checked into the source code repository, "crashes" which receives automated reports when a user's copy of the application crashes. Finally there is a little used "announce" list and sundry lists which broadcast activity in the Trackers, created since the participants wanted to get an easier understanding of cross-venue activity. BibDesk uses two Trackers: Bugs and RFEs (Request for Enhancements). The project has not adopted many of the new services offered by Sourceforge over time, choosing not to use Forums, the Task manager, Skills listings or Services (a marketplace for paid support and development). The project did switch from CVS to SVN and does accept donations through the Sourceforge system, although there are very few and they are not a topic of discussion on the BibDesk mailing lists.

Beyond the Sourceforge tools, BibDesk has a website with a linked Wiki, which has grown in use and usefulness over the years (despite a slow and unstable start). Finally there is hosting for daily releases, which is provided through a developer's private resources and built automatically on a developer's personal computer. The project has never used web forums, or real-time chat systems such as IRC or IM. As far as I known, developers very rarely email each other "out-of-band"; I have done so only twice: the declined suggestion of a face to face meeting and once to give moral support to a developer facing a particularly argumentative, stubborn and rude user making unreasonable requests. There is good evidence that all the developers are using the Integrated Development Environment provided by Apple, known as XCode and Interface Builder.

3.4 BibDesk as an experience

While the previous section presented a high-level and abstract view of BibDesk the project, this section presents an intimate, personal view of BibDesk. The section is organized in three chronological parts: initial contact, peak contribution and ongoing participation. Through these runs two themes. The first is the relationship between venues and depth of participation and the second is the burstiness of work. Finally two episodes of my work in the project are presented in greater detail.

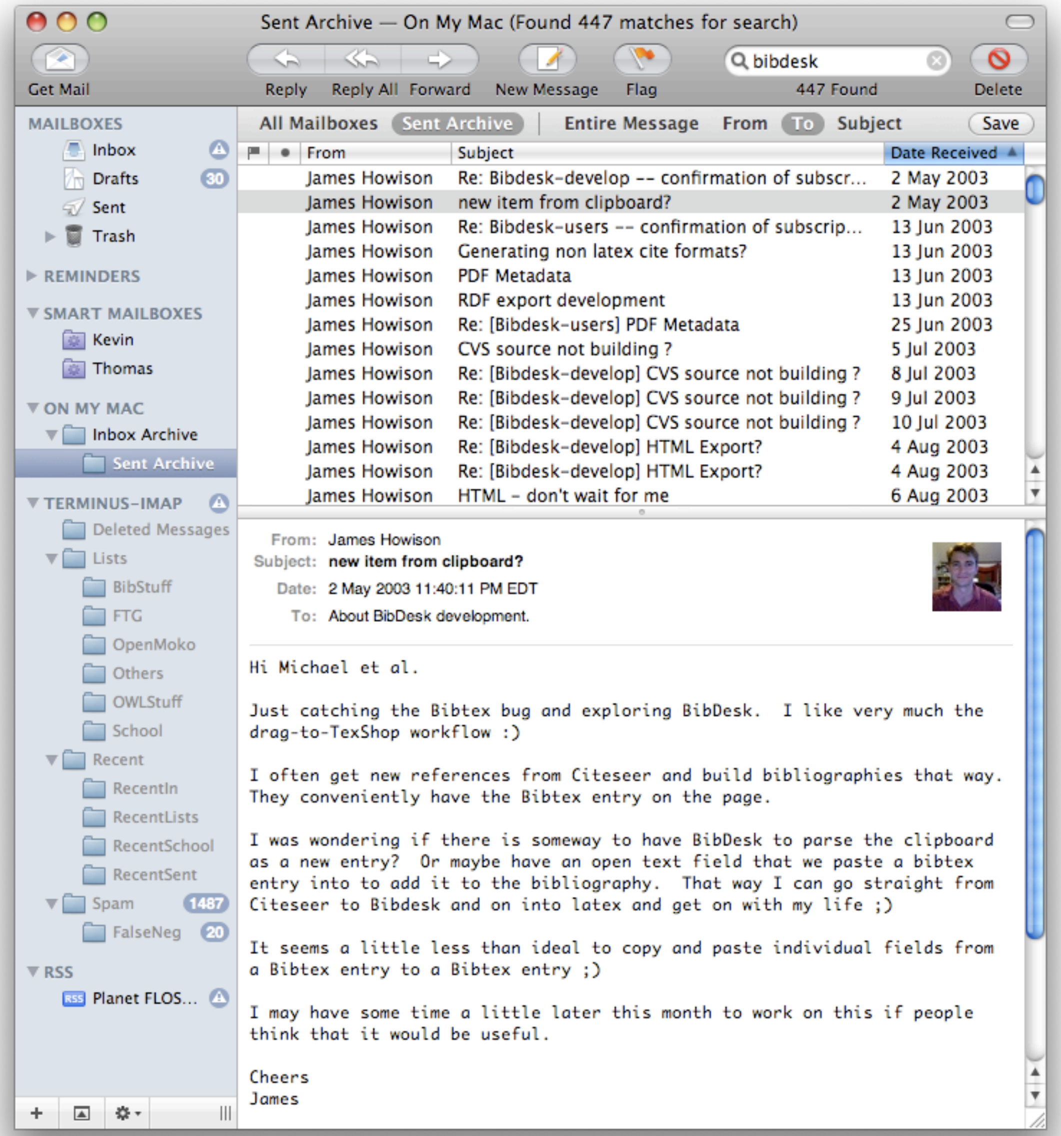
First Contact

My first contact with the project was through the application itself; downloading and using it in my day to day work. I remember being being literally delighted with the software. It is well integrated into the Mac OS X system, which is itself aesthetically pleasing. The application was functional for editing BibTeX files, although nothing like as feature-full as it is today. While the application may not initially appear to constitute contact with the project, per se, I believe that the experience embodied in the application is important. Superficially one encounters relevant participant names in a number of incidental places during download: the Sourceforge page, the release notes and in the "About This Application" credits box. Furthermore actually using the application is the primary shared activity of the project; descriptions and discussions of the application, its features and use (as opposed to development), dominate the life of the project, even for participants who are writing code.

From here my next contact with the project was through the mailing lists. My personal email records that I first joined the bibdesk-develop mailing list (2 May 2003), then the bibdesk-users mailing list (13 June 2003). I joined the bibdesk-develop mailing list because I wished to speak there, not with the intention of lurking. My first post to that list was on 3 May 2003 and was an inquiry gently seeking a feature which proved to already be available. I began the email with a personal salutation to the project founder and ended it with a vague offer of assistance.

My second contact was with the users mailing list where I summarized my positive experiences with BibDesk and presented three associated ideas I wanted to pursue (I

Figure 3.2: My email client records my subscription to the BibDesk lists and my first few contacts



From	Subject	Date Received
James Howison	Re: Bibdesk-develop -- confirmation of subscr...	2 May 2003
James Howison	new item from clipboard?	2 May 2003
James Howison	Re: Bibdesk-users -- confirmation of subscrip...	13 Jun 2003
James Howison	Generating non latex cite formats?	13 Jun 2003
James Howison	PDF Metadata	13 Jun 2003
James Howison	RDF export development	13 Jun 2003
James Howison	Re: [Bibdesk-users] PDF Metadata	25 Jun 2003
James Howison	CVS source not building ?	5 Jul 2003
James Howison	Re: [Bibdesk-develop] CVS source not building ?	8 Jul 2003
James Howison	Re: [Bibdesk-develop] CVS source not building ?	9 Jul 2003
James Howison	Re: [Bibdesk-develop] CVS source not building ?	10 Jul 2003
James Howison	Re: [Bibdesk-develop] HTML Export?	4 Aug 2003
James Howison	Re: [Bibdesk-develop] HTML Export?	4 Aug 2003
James Howison	HTML - don't wait for me	6 Aug 2003

From: James Howison
Subject: **new item from clipboard?**
Date: 2 May 2003 11:40:11 PM EDT
To: About BibDesk development.

```
Hi Michael et al.

Just catching the Bibtex bug and exploring BibDesk.  I like very much the
drag-to-TexShop workflow :)

I often get new references from Citeseer and build bibliographies that way.
They conveniently have the Bibtex entry on the page.

I was wondering if there is someway to have BibDesk to parse the clipboard
as a new entry?  Or maybe have an open text field that we paste a bibtex
entry into to add it to the bibliography.  That way I can go straight from
Citeseer to Bibdesk and on into latex and get on with my life ;)

It seems a little less than ideal to copy and paste individual fields from
a Bibtex entry to a Bibtex entry ;)

I may have some time a little later this month to work on this if people
think that it would be useful.

Cheers
James
```

return to this below). The response from the project founder was positive, and he gently directed me to the code, including a specific file and line. I felt both welcomed and challenged, but not overwhelmed.

Fired up, I was then 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. Anonymous access is read-only and allows projects to meet their source code distribution obligations, allowing outsiders to build, use and learn from their code, while still allowing the core developers to avoid potential vandalism and maintain the one real control point. In the case of BibDesk write access was granted without much ceremony, usually after a potential developer had provided one or two patches.

In simultaneous reading about FLOSS projects I had encountered the notion that CVS "should always build" and this was brought home immediately, since I was unable to build the software and therefore could not hope to begin the private, local cycle of change, build, test that came to be familiar as a FLOSS developer (see Figure 3.4). I decided that helping to fix this situation would be my first real contribution, so I detailed my experience and difficulties in an email to the develop list, pointing out exactly which changes I'd had to make in my local infrastructure and the source code to have BibDesk build.

In this process I encountered my first experience of the layering that is core to FLOSS development. At its core BibDesk is a relatively thin layer which provides a user interface and glue for three sets of libraries. The first are the Cocoa libraries provided by the Mac OS X operating system, which deal primarily with on-screen user interface aspects such as the central table-view but also provide an interface to lower level operating system components, such as File Dialog boxes and programatic actions such as File Save. The second core library, called *btparse*, is an open source library, written in C, for parsing and writing the BibTeX file format. The third set of libraries are those provided by the OmniGroup, which complement and extend the basic Cocoa services, making application development far easier. I had difficulties configuring my build environment, using the Apple provided IDE to build and use the OmniGroup libraries. The email thread I began helped the project properly generalize this environment beyond the founder's computer. This is not uncommon in FLOSS projects: for years the default settings for compiling the Linux kernel were notorious for being specific to the rather underpowered personal computer of Linus Torvalds (the founder and central author of Linux). The email exchange about building provided the support I needed to get things setup properly, although I did not stop experimenting as I waited for a reply, understanding that they might be a while in coming and I might be able to resolve the issue myself in the meantime.

With my development environment eventually established I was able to write the very small patch that the project founder had suggested in his reply to my second contact (it was simply a fix to a missing line-break, just a step above a typo). Interestingly it was clear at the outset that the project founder would have been able to make this fix himself, in a fraction of the time it took me, but that he restrained himself. This fits with the practitioner-scholar recommendations of "The Cathedral and the Bazaar" to leave some "low hanging fruit" (Raymond, 1998). I was thanked in the CVS check-in note and the release note for this patch.

Following up the building issue, I wrote a Perl script which eased the process of setting up a fully functional development environment. The project founder granted me CVS access to check in changes to this script myself. The script did not actually interact with the project source code, but simply automated a series of CVS commands to get parts necessary for building into the right places, as well as provide some documentation of needed build settings. I maintained this script for over a year, until a reorganization of the CVS made it redundant.

During this time I simultaneously became an "active user", by which I mean that I used the application, reported bugs and helped to answer questions on both the users and develop mailing list. I was also actively building and testing the code from CVS, particularly as the project headed towards another release. I was never a particularly active developer; my understanding of the Cocoa frameworks and Objective C (the language of the project) was quite limited and the learning curve quite high, despite the intermittent support of the other developers. I noticed that the more productive developers were intimately familiar with both of these, and each had multiple other open source and non-open source projects that used them. Nonetheless, from time to time, I continued to make small contributions to the BibDesk project, such as adding a Container column (detailed below) to the table view and fixing help files. Over time I came to understand the Trackers as a place for long-term issues and began to use them more.

Specific Episodes

In this section I outline two episodes of my involvement as a peripheral developer in the BibDesk project. The first was a successfully completed episode in which I added a new column to the main application view. The second was an ultimately unsuccessful episode[1] where I sought to add parsable metadata to the PDFs managed by BibDesk. In addition I discuss a long-term situation with regard to a putative BibDesk 2.0.

Successful work: the Container column

In Figure 3.1 you can see the results of one task I undertook as part of the BibDesk project. The second-last column from the right has the header 'Container'. For

[1]http://sourceforge.net/mailarchive/message.php?msg_id=8710bc3c77e48685b550
adac0e39315a%40syr.edu

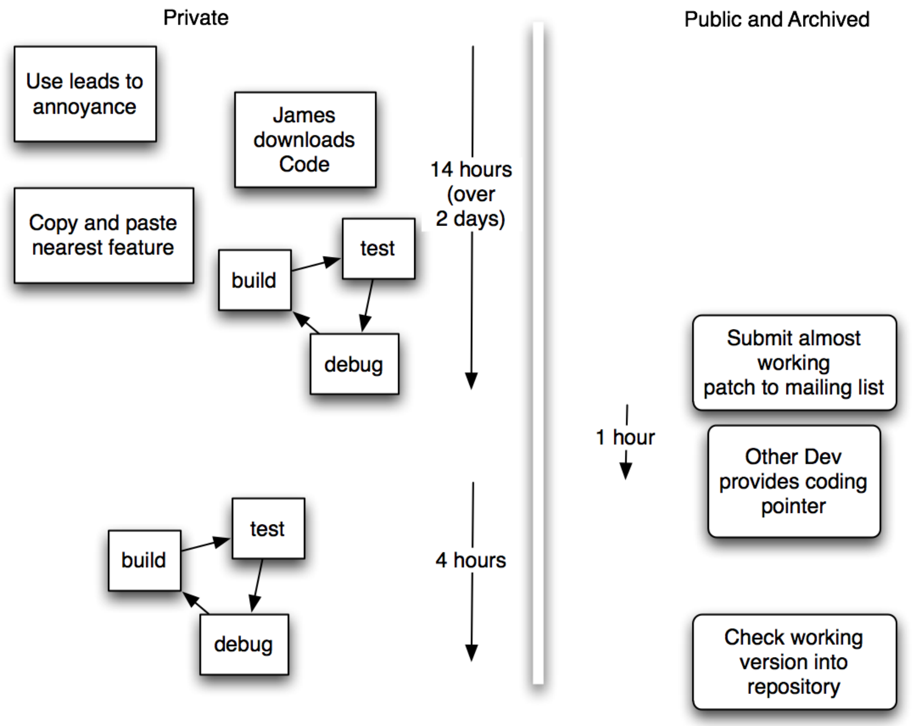
articles this displays the Journal title and for Conference Proceedings this displays the title of the conference or proceedings; even though these elements are stored in differently named field in the underlying BibTeX (journal and booktitle, respectively). The purpose of this column is therefore simply to save space, so that a user doesn't have to have the the often empty journal and booktitle columns visible.

I undertook this task in April 2005, after having been an intermittent developer and active user for over a year. I was motivated purely out of personal annoyance. At the time I was using a 12″ laptop and had limited screen space. I was trying to get an overview of publications for a readings and research class and found the table structure to waste space.

I worked on this largely 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 codebase. 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 and replied with comments on how to fix the sorting issue and said he thought it should be committed. After 4 hours further work, I resolved the sorting issue and committed the patch.

The flow of work is depicted in Figure 3.3. In total I estimate that it took about 20 hours of work, which was spread over three days. I worked on this mostly at night. The time was spent in Project Builder, Apple's development IDE; with the Apple library documentation and in very short cycles of change, compile, test familiar to any programmer. I was very much "in the moment" and fully occupied with this problem, and spent perhaps 12 hours trying to fix the sorting issue. I finally sent the patch to the list, admitting that I needed help, at 9:52pm. The project founder, also a student but based on the west coast of the US, responded at 10:24pm including his recommendations for fixing the sorting issue, and one or two improvements with memory management (never a strong point of mine).

Figure 3.3: The flow of work in building the Container column, including public and private activity.


Private
Public and Archived
Use leads to annoyance
James downloads Code
14 hours (over 2 days)
Copy and paste nearest feature
build
test
debug
Submit almost working patch to mailing list
1 hour
Other Dev provides coding pointer
build
test
debug
4 hours
Check working version into repository

I then worked for the next 4 hours to implement his suggestions, checked the feature into CVS and replied to his email at 4am. I had not wanted to reply immediately to indicate that I was working on it, since I wanted to demonstrate my competence by replying that it was all complete.

Failing to add metadata

One of my original intentions with BibDesk was to add bibliographic metadata directly to the PDF of academic articles. This would enable PDFs to be managed "like mp3s". Simultaneously I was working on this idea as a research practicum, which resulted in a conference publication outlining the idea (Howison and Goodrum, 2004). This feature was one of the three ideas which I brought up in my second contact with the project email lists.

Similarly to the Container column episode this was driven by motivations of personal annoyance, although this was more abstract and grandiose: it was part of a research agenda for improving academic workflows. I had to work harder to nail down my real intentions, discussing them with the project and with my academic colleagues. When the time came to write the code I faced a serious problem which blocked my ability to implement the feature. The idea was quite simple: one would simply write metadata into the PDF, then later read it out. However I was not skilled enough to write the code for writing metadata into the PDF by myself; it turns out to be a complex, check-summed, file format that is not amenable, for example, to adding data to the start of the end of the file (as is done with the ID3 tags for MP3s). The other BibDesk developers agreed that the feature sounded useful, but not particularly enthusiastically; it remained my intention rather than becoming a group goal. They had useful suggestions but also did not have familiarity with the PDF file format.

I realized that I would need to stand on the shoulders of giants, and so I began to search for a library that could provide these low-level features; my task would be to stitch them together at a process and UI level. Unfortunately I was never able to find such a library. Adobe, the makers of PDF, did provide a library that offered at least part of the needed features but licensing issues made it incompatible with open source software. My inquiries about changing this met receptive ears at Adobe, but action was difficult to achieve. None of the other PDF libraries offered flexible enough operations on the PDF file. I did investigate the PDF file format myself, but it was too complex for me to resolve. I was worried I would invest significant amounts of time learning the file format, only to be blocked by another, unforeseen, issue further down the line.

So I set the implementation aside for a time. After the conference article was published (and blogged) I received many communications about the implementation it promised, including from Nature, the scientific publishers, who wished to incorporate it into their workflow. Partly for this reason I returned to this task from time to time checking to see whether new libraries had emerged that made it easier. I also developed a set of contacts who were also interested, from similar FLOSS projects, such as JabRef, and we followed the issue with interest; but none had the skills or time necessary to implement the needed functions. It is only in the past year that libraries have emerged with the needed capabilities, and the feature was eventually implemented in JabRef. Unfortunately I haven't had time to port it to BibDesk yet.

BibDesk 2.0

The final episode I want to highlight was a long running period in which the intention of the group was to release BibDesk 2.0. As of October 2008 the current version is 1.3.18, which is to say that BibDesk 2.0 never emerged. The idea was floated by the project founder just prior to BibDesk achieving 1.0 status; which was itself prompted by a desire to "tie-up loose ends" prior to an effort to build BibDesk 2.0. The effort began in the same way as BibDesk 1.0 and Skim.app: as a private project of the project founder which was eventually moved into BibDesk's public CVS, as a new CVS module. It took advantage of a set of new libraries released by Apple in their 10.4 operating system, known as Core Data. 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. BibTeX was to be replaced as the underlying file format, and instead be simply one export format among many.

However the project never caught hold and rather than switching work to the BibDesk 2.0 module 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). I managed to build the 2.0 project a number of times, but the functionality was low compared to the 1.0 module and therefore I could not adopt it for day to day work.

Instead work has progressed on the BibDesk 1.0 module in smaller steps than that imagined for 2.0. Nonetheless the developers eventually achieved most of the features planned for 2.0; such as a vastly improved group system, a very flexible templates system and 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 existing structure of the 1.0 software and even the reliance on the plain text BibTeX file format.

3.5 Findings

This section presents findings related to the three sensitizing concepts of the study and the overall research questions of this dissertation.

Motivation

Participant observation enables the researcher to report on issues that are hard to capture in other ways. This section describes my introspective understandings of issues related to motivation in the BibDesk project. While I cannot speak for the other developers I have not seen behaviors or discussion which would tend to contradict these findings.

I was motivated to participate in BibDesk by two over-arching drivers: a desire for software that served my needs and a desire for learning. As described above BibDesk was useful to me in my day to day practice, and during that practice I came to truly understand the phrase “to scratch an itch” (Raymond, 1998). Day to day use of the software reminds the developer of elements which do not work as they would like; the elements literally get in the developer’s face, interrupting their workflow and continually starting the developer down the path of thinking about how to fix them. It was my experience many times to be taken from my academic writing work into jotting down thoughts or poking at the BibDesk source code to think about how to resolve an aspect of BibDesk that was showing up during my work. This continual prodding in the course of daily life is an extremely effective motivator, and the instant gratification of available source code is not to be underestimated as an involving and motivating factor. Perhaps one could even say that procrastination from “real work” created opportunities for working on BibDesk.

By contrast my motivation for learning was more periodic, surfacing irregularly and driving activity only in patches. Firstly, and importantly, one must acknowledge that I was motivated by a desire for a field site and for academic career reasons. Yet this motivation faded into the background, surfacing strongly when my other research or reading on FLOSS intersected with experiences in the BibDesk project. From time to time I was motivated to learn about the Apple libraries, Cocoa and Core Data, but these are vast topics and this was not greatly sustaining. My failure to learn sufficiently to implement the PDF metadata supports this finding. For me, at least, while learning was a motivator it was less important and less constant than the daily prodding of software use.

Motivation also played a substantial role in day to day task work in the BibDesk project; the organization of work affected my motivations and vice versa. For example it was often the case that a small piece of successful work, such as my Container column, lead onward to other successful, small, pieces of work, in a reinforcing cycle. For example, in my completion reporting email[2] from the Container episode, I wrote "BTW, are we meant to be using [pub stringByRemovingTeX] for display in the tableview? ..." which indicates that my attention had now been caught by a new issue in the course of the work.

By contrast if I was blocked in my activities I found this consistently de-motivating. For example if the Sourceforge CVS system was down—sadly a too common occurrence for a period of time—I lost that minute to minute impetus to try something in the code, similarly if the CVS tree was in an unbuildable state (which happened, but only rarely).

The BibDesk 2.0 episode illustrates this and one other finding. The entire development group encountered this during the BibDesk 2.0 push, as small feature extension work slowed while the group waited for the 2.0 framework. The group's plans in this period were far more explicit than at any other time: the project founder was to build the 2.0 framework and meanwhile the other developers were to tidy up bugs in the 1.0 module. It was assumed that the project founder would 'come through with the goods' and so others did not 'hack on' the 2.0 plans, deferring their feature expansion plans. As the time to a working 2.0 extended, eventually momentum for features overcame this and individual developers, while paying lip service to 2.0, began to add features to the 1.0 module again. Eventually 2.0 came to only be mentioned in passing, as the 1.0 module advanced past features originally planned for it.

Interaction during a work task was motivating, particularly if it came quickly, while my attention was still focused on the topic. Sometimes this took the form of a quick, offhand suggestion which eased one's progress through a work task. Yet it was interesting to me that a lack of feedback was not particularly demotivating,

[2]http://sourceforge.net/mailarchive/message.php?msg_name=abfcbca38becde478dcceac 8412f7095%40syr.edu

certainly not as much as being blocked in my work. A response was read as positive feedback, but the absence of a response is harder to read; it might simply indicate that the person is sleeping, on vacation, hacking hard on another project or otherwise attending to their "real life".

Coordination and Dependencies

My experience and observation of BibDesk considered in the light of my growing understanding of the coordination and dependency literature gives rise to three findings.

Successful work was shorter

Firstly, my successful work tasks, and many tasks I saw completed, were relatively short, extending only over a matter of days, rather than weeks. Conversely my failures, and others I saw (including the 2.0 push), were longer, taking place over weeks, even months. On one hand this is obvious; uncompleted tasks continue, yet the eventual outcome of longer work was much more likely to be abandonment than completion.

Absence of planned reliance

Secondly, my work in BibDesk was my own and was accomplished without planned reliance on other developers. It was never the case that tasks were allocated, or that I made a bargain beforehand with another developer to work together to complete a bug fix or implement a feature. I do not mean that my work was not improved or sped up by interactions with others; it often was. Indeed many times my rough code was integrated and polished by others, just as I sometimes fixed typos etc. Yet these interactions were spontaneous and unplanned, welcome when they occurred and basically unnoticed when they did not.

Of course it is true that I relied on code provided by other developers and every single activity in the project relied on the code originally provided by the project founder and all the code in the repository. Yet this did not feel like reliance on that person; after all they could not remove the code and while it was sometimes helpful to have the author explain code in email, given enough time one and a debugger one could usually figure out how it worked.

Deferring work as a strategy

Thirdly, I came to understand that, given the constraints of time and skills, much work in BibDesk was deferred. By deferred I mean that it was eventually accomplished, but sometimes months or even years after it was first proposed, when changes in circumstances provided the time, energy or ability. My earliest discussion[3] on the BibDesk lists, in June of 2003, speculated about being able to subscribe to and automatically parse the publication lists of fellow academics. The project founder and I discussed the desirability of this; indeed one of BibDesk's less used features was the ability to publish a set of publications as an RSS feed (a semi-structured machine readable format used for blogs). Yet implementing the ability to subscribe to publication lists required quite a number of things to fall into place; from giving BibDesk the ability to read web-pages, to finding standard enough formats to providing a suitable UI.

The project founder deferred this feature for over 3 years, until the very start of 2007. He returned somewhat unexpectedly after a long hiatus from BibDesk development and provided a patch that implemented the ability to subscribe to pages with associated BibTeX, including Google Scholar results. In the past year another developer had implemented the idea of Groups of references, including those grouped by author, or keyword and so on. The project founder used this infrastructure to layer

[3]http://sourceforge.net/mailarchive/message.php?msg_name=7394DD78-A02E-11D7-AFC1-0003931E45D0%40mac.com

his "web group" in just a few days of coding; finally realizing the core of a desire we'd discussed years before. As he provided this code he commented,[4]

> *I just hacked together a new group, the "Web" group - the idea is if BibDesk is iTunes for your refs, then the whole web is the music store . . . 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. Thanks!*

Deferred work is not work abandoned. I was forced to defer my plans for PDF metadata, but the ideas remained available, on the mailing list, in publications and on my blog. Eventually someone else did let me know that a library was available that could do the job, and they, in fact, implemented the feature for a different project (JabRef).

At a shorter-term level the Trackers play an important function in regard to deferred work. Trackers provide a single, persistent web page for each issue. Provided the issue is not closed by an administrator, this allows the project to remember the past and allows comments to build up over time. The comments are sometimes 'votes' from others encountering the same issue, they are sometimes comments from developers regarding attempts at solving it, or information about how other projects solved such problems. The project, therefore, was not particularly concerned about "outstanding" RFEs or even bugs (as long as they didn't result in data loss); these were memos on the board, saying 'we'll get back to it' and they often did.

[4]http://sourceforge.net/mailarchive/message.php?msg_id=DF0FB757-56BA-45D7-A1EA-262EB7A5B3DC@mac.com

The experience of organizing: Discovering episodic individual work

The final finding to be reported here relates to the question of how to represent organizing in the FLOSS context. The process of my integration with the BibDesk project, as related above, was one of rapidly expanding interaction which moved progressively into differing project venues; from the application, to the mailing lists, to anonymous CVS and, longer term, to Trackers.

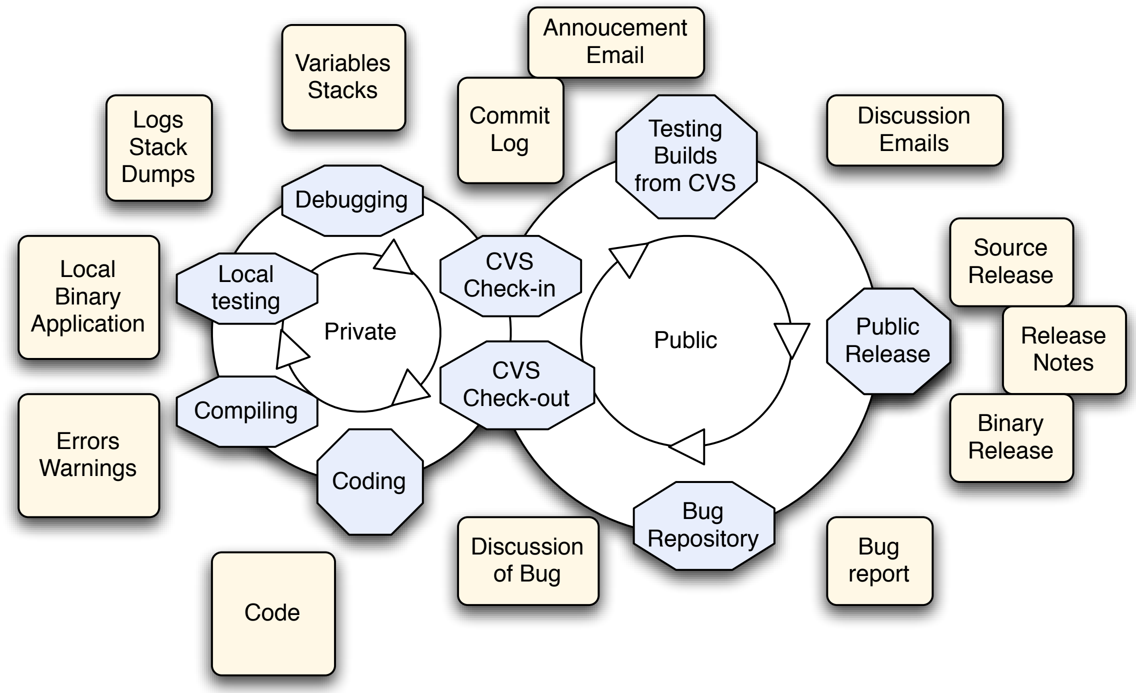
I came to appreciate many aspects of this infrastructure, I highlight only three of them here, since they relate directly to the theory development in Chapter 7. They are: selective transparency, opportunistic synchronicity and the largely individual manner in which work unfolded in episodes. These are truly socio-technical phenomena; they are not prescribed by the technology, rather social practices arise which exploit—and are shaped by—features of the technology in particular patterned ways, some of which are only clear as a result of participation.

Selective Transparency

It is often stated that FLOSS projects are completely transparent, everything is laid bare for any observer to witness (e.g. Harrison, 2001). In some important ways this is true, yet the transparency of the archives is selective. It is true in that FLOSS projects make available far more records of their life than do other types of work. They do so in ways that are a side-effect of their collaboration, their publication is automatic and doesn't require additional work.

Firstly, not all of the life of the project ever makes it to the archives; some never does. Secondly, different venues record different aspects of life and are more difficult for the observer to penetrate with their understanding. Finally, that which does is

Figure 3.4: A great deal of the life of a project is private and not recorded in the archives; this figure shows interconnecting private and public activities and the artifacts they create.



archived in particular ways, using archival technologies which impose themselves on the reader, simultaneously exposing and hiding parts of the project's life.

The archives of a project record only what the participants in the project wish to make public; as a participant one is quite conscious of what is public and what is private. This can been seen on many academic mailing lists when a participant accidentally replies "in public" rather than "in private" and quickly publicly regrets this. Figure 3.4 shows two interacting cycles in development—one public, one private—and the artifacts they produce.

Much of my actual working time on BibDesk was not at all visible to the other developers or project participants. They are able to see emails, tracker posts and CVS checkins if my work is eventually successful, but not the many aborted efforts along the way. Further much of my time spent working on BibDesk was never visible, since it did not result in working code. Final check-ins did not reflect the difficulty of preparing them.

Similarly much "project time" is spent reading or watching the various project venues, an entirely private activity, revealed in records only if one replies. As Lakhani and von Hippel (2003) confirm, over 98% of time spent in user support forums is reading, rather than writing responses which become visible.

Not all project venues are equally available; some are more difficult to penetrate and to understand than others. Consider my own trajectory. I began with the application and release notes, progressed to the develop and user mailing lists, then to read-only CVS and to Trackers. The mailing lists, as email, are the most familiar to outsiders who are likely already participating in many other lists. For more recent projects, and younger potential participants, Forums provide this familiar entry point.

CVS, by contrast, is obscure and technically intimidating; I still do not understand all the ins and outs of some of the operations, like branch merging. Yet CVS commits and diffs are commonly read and understood by developers, using both the CVS log messages, the lists of changed files and edits to special files, such as a project README file.

Finally archives are presented through technologies that subtly alter their interpretation, making being part of a project quite different from observing it in retrospect. For example, I read the BibDesk lists, including CVS log updates, in an email client, not in an email archive. This has at least two impacts. The first is temporal; email arrives and is read at different times. This allows it to function as an attention indicator, telling attentive project members that another participant is online and available now. Secondly, email as viewed in a mail client has a limited time horizon; eventually older threads are pushed off the visible list of emails by newly arriving ones.

One crucial result of these processes is that evidence about work is scattered throughout the project's archives. Work that proceeded cohesively for the participants, bound together by time, attention and topic, is presented differently in the archives, artificially separated by venue, obscured by archival techniques and more or less stripped of its temporal cohesion. As a participant one experiences work as a whole; as an archive reader one must work extremely hard to reconstruct this work.

Seeing work through archives

It is important to separate three structures of inter-relatedness in the archival record. These structures are Discussions, Tasks and Sessions. By Discussions I mean the patterns of interpersonal call and response; visible most clearly within a single tracker item or in email threads, where they are less well preserved. Here a sequence of discussion is evident, with contributions crafted to respond to previous contributions in a way analogous to conversation. Discussions are almost always limited to individual venues, rarely do they cross venue and thus archival lines (an uncommon exception is moving a discussion from a user list to the developer list through forwarding).

The second structure is that of Task, by which I mean a pattern of evidence which is linked by having been caused by work towards a specific change in the application. Tasks result in meaningful changes to the shared output of the project. These patterns have no respect for venue and archival boundaries because participants are focused on their private cycle of development and turn to different venues as that private cycle dictates.

Overlaid on both of these is a third pattern derived from the fact that software work occurs when participants are sitting at their computers and that people, even open source programmers, don't sit at their computers all the time. Rather participants participate in Sessions, temporal periods in which the participant's attention is turned to the project, whether that be reading and responding to project email, actively pursuing a programming problem or reading (and perhaps editing) project documentation. The length of time that participants spend clearly varies. For me this varied from a quick 10 minute scan of project lists, catching up on interesting threads, to Sessions approaching 8 hours, ended only by the real life, sleep and PhD coursework.

On top of this, again, is laid a second type of temporal pattern caused by the waxing and waning of a participant's involvement in the project over periods of weeks and months. The sources of these patterns are not as clear-cut or universal. For me it was a question of what was occurring in my real life, whether that be coursework, research work or travel. Further even when I had relatively normal availability of near-free time, my commitment to the project varied. I think this was because my use of BibDesk varied, peaking during writing periods, during which even if I resisted instant gratification of heading to the code, I built up a set of annoyances or visions of changes I hoped to make or encourage others to make.

Opportunistic Synchronicity

The participants in BibDesk did not plan to be co-present, to align their working sessions temporally; yet from time to time the core developers seemed to overlap and to take advantage of this situation. In my own work, as related above, when I reached out via email I found quick responses to be motivating and to create a sense of companionship, but not reliance.

The ability to take advantage of such opportunities and switch from asynchronous communication to near-synchronous communication is supported by the temporal flexibility of email, in particular. As a push mechanism, delivered to inboxes which most participants are automatically checking regularly while working, sending an email on a list functions not only as a communication channel but also as a presence indicator. A quick response indicates that you too are present, have time to consider the project and are interested in this particular Task. In this way a Discussion can shift from an asynchronous mode where emails are fairly long, more akin to journaling than conversation, to near-synchronous discussions proceeding in "turns" much closer to face-to-face conversation. Crucially these shifts occur without requiring the participants to change media, say to a telephone or IRC, and are archived in the same way, allowing those not present to catch up during their next Session.

Yet because there is no pre-commitment to co-presence there is only a weak expectation—perhaps a hope—that a quick response will catch the other participants still at their computers, still with their attention turned to project venues. There is, perhaps, a hope of quickly sharing one's work and getting input, but there is little disappointment or discouragement if it does not eventuate; after all one does not necessarily know why the others aren't responding, they may well just be getting some well deserved sleep. Near-synchronous, focused attention is "nice if it happens", but is simply additional and enhancing, not vital to BibDesk's work.

Episodic individual work

For non participants, these features together obscure the manner in which work unfolds in BibDesk. Work is accomplished during individual's Sessions, and driven by both Tasks and Discussions. Work is temporally and topically cohesive to the participant, yet leaves traces scattered, and altered, throughout different archival venues. Interpersonal interaction is almost always unplanned, often even incidental or accidental, supported by the flexibility of the project's media use, but not a fundamental requirement of participation in a project. Work does not stop if interaction does not appear. My understanding of the project was that very little, if any, work involves more than a single developer doing programming work, although other developers may provide incidental support as the main developer works through their self-chosen task. Other types of activity in the project, like user support, were similarly skewed towards contribution from individual developers, rarely requiring input from more than one developer.

The findings discussed above summarize the learnings from the BibDesk participant observation that are relevant to the research questions of this dissertation and are summarized in Table 3.3, below.

Pilot Study: Types of Collaboration

To confirm that my understanding of episodic individual work in BibDesk was substantial and not simply an idiosyncratic reading, I re-examined the archives of the BibDesk project for an entire inter-release period, drawing on my emic understanding of the organization of work.

Manually, I collected all the archives of the BibDesk project for a single inter-release period. I chose BibDesk 1.1.2 which was released on 1 July 2005. I had spidered Sourceforge to obtain the mailing list, but I manually collected all other venues, copying and pasting Tracker items, for example, into a working file.

My primary goal was to reconstruct the activity that participants performed during the period, by resorting the archives from their media-based locations and grouping them as episodes linked by the activity that seemed to give rise to them. I did not restrict my interest to episodes resulting in changes to the codebase, and so also included episodes of user support and non-production focused developer discussion.

I began with Trackers that were closed during the inter-release period, since closing a Tracker indicates completed work (or a rejection of work), seeking relevant evidence in CVS and on the mailing list. I then turned to CVS checkins, working with both the log messages and, from time to time, the code changes themselves. As I discovered relevant communications I cut them from a working copy of the original archive, and pasted them into a file for an episode. Finally I turned to the mailing lists, developer first then user, and tried to group the remaining messages.

I then worked to summarize the documents, turning them into a single sentence description of work, separated by a vertical distance in the file, to approximate their separation in time. I then turned the episode into a sequence of initials indicating the actors identity, separated by a gap (. . .) if the event occurred more than 8 hours after the last. I then identified the overall role of the actor as either a developer or a non-developer, based on my personal understandings of their long-term role. I did not base that classification on their particular role in the Episode, nor on whether they had CVS check-in privileges or not.

Table 3.1: Sample BibDesk Episodes

#	Name	Kind	Pattern	Devs	Synch?
1	bst macros	RFE-fixed	U1-. . . -AM-AM-. . . -AM-. . . -AM-. . . -AM-. . . -CH-AM-CH-CH-. . . -CH-. . . -CH-. . . -AM	2	Yes
64	volatile	bug-fixed	AM	1	No
34	preview crash	bug-not fixed	U1-AM-. . . -CH-AM-. . . -AM	2	No
38	macro defs	feature imp	AM-AM-AM	1	No
53	german capitals	user support	U1-AM-U1-U2-AM-U3	1	No

In this way I found 60 episodes, with summary strings as shown in Table 3.1. I used this data to confirm the understandings outlined in the chapter above.

This case study fell early in my theory development process, when my concepts and questions were not as well defined as they are later in this dissertation. Further the manual nature of the archive organization did not lend itself to recoding. Therefore the quantitative summaries can only weakly capture some of the understandings outlined above and fully developed later in this study. Nonetheless, as shown in Table 3.2 all but one of the episodes involved one of the three identified developers, 58% involved only a single developer and 40% involved more than one developer. Fully 46% (29) of episodes had near-synchronous (within 8 hours) interaction between two participants, but only a few of those involved near-synchronous interaction between two developers, only 13% (8) overall.

Table 3.2: Patterns of developer involvement in BibDesk

Pattern	Count	%
Zero developers	1	2%
Single Developer	35	58%
Multiple Developers	24	40%
Total	60	100%

It is important to note that this analysis is different from the more structured and focused analysis of Fire and Gaim to be reported in Chapter 5; it is primarily a systematic check for my participant observation results reported above and only secondarily a pilot for the work to come. Firstly all forms of activity in the project were included, not just activity resulting in substantive contribution. Secondly, the roles of the participants were defined overall, not based on their actions during an episode. A later review confirms that often one of the three participants identified as a developer—based on their lifetime contribution to the project—was not acting as a developer in the sense of producer within some of the multiple development episodes reported above.

3.6 Case study Conclusions

The work in BibDesk was not alone, but neither was it like teamwork. “Working in company” seems a closer description; it was companionship rather than co-labor. My experience and my pilot systematic archive study indicate that the work in BibDesk was, far more often than not, individual work, but that it took place with a supportive audience, albeit one that was often otherwise busy but could, from time to time, come through with timely support.

Table 3.3: Participant Observation Findings

#	Finding	Justification
	Motivation	
1	I was motivated primarily by my day-to-day use of the software	Introspection
2	Participation was sometimes stress-relieving procrastination from “real-work”	Self-critique
3	I was also motivated by learning but this came and went	Introspection
4	The experience of work affected motivation (success and rapid interaction was motivating; failure was demotivating)	Introspection
	Coordination and Interdependency	
5	Successful work, my and others, was shorter, days not weeks	Observation
6	There was almost no planned reliance on other people’s time (no allocation of tasks, few pre-announcements, few requests)	Observation
7	Helping behavior was unplanned and opportunistic	Obs. & Intro.
8	Difficult, but desirable work was deferred and sometimes made easier by changes	Observation
9	Trackers, due to their longevity, were used as memos for deferral	Obs. & Intro.
	Experienced Organization	
10	The life of the project was given coherence through three types of structures: – Sessions: short periods of time at the computer where attention is on the project – Discussions: Call and response patterns within a single venue (e.g. Email threads) – Tasks: Work directed towards a specific goal of changing the application	Introspection
11	Tasks left documentary evidence throughout the venues of a project	Archive study
12	Different venues have different archiving affordances which affect interpretation	Archive study
13	Much individual activity, especially during development, is not archived	Intro. & Arch.

My experience in BibDesk also revealed the layering of software, as well as the common deferral of work. My motivational introspection was in synch with the literature on FLOSS motivations reviewed in the previous chapter, but introduced the suggestion that motivation is affected by blockages in production.

As with any case study, however, and particularly one that draws so heavily on personal experience, a scholar must be conscious of the possibility that one’s experience is particular. I tested some of my own experiential understandings within BibDesk with a systematic pilot archival study of the work of the whole project, but the possibility remains that the practices and experiences in BibDesk are unique to that project and therefore are not, as this dissertation argues overall, linked to the socio-technical nature of FLOSS work. Therefore the dissertation now turns to efforts to focus and replicate these findings before formalizing the combined findings in a socio-technical theory of FLOSS organizing. First however, we return to the literature to assist in the design of the replication study and to deal with the experienced episodic organization of BibDesk.

Chapter 4

Literature Review II

This chapter introduces literature which updates the Input-Process-Output framework introduced in Chapter 2 to cope with long-standing and episodic teams, and to provide a solid conceptual basis for attempting to replicate key findings through archival case studies. As is usual with qualitative and experiential research, much of this literature was encountered during the participant observation and already influenced the findings. It is presented here for the sake of narrative clarity. It is hoped that the previous description of the BibDesk study will provide a concrete grounding for understanding the relevance of the literature, as it did for the author.

In the overall structure of the dissertation-as-document this chapter updates the IPO approach to teamwork and shows how these developments in the literature support the findings in Chapter 3 and point to an empirical strategy for replication. Most importantly, however, this chapter helps develop a solid conceptual basis for attempting to replicate key findings through archival case studies, by reviewing the literature on coordination and interdependency.

4.1 Time and the IPO approach

One of the key findings in Chapter 3 was the manner in which organization was experienced as an overlapping set of Tasks, Discussions and Sessions. It was a far more complex, yet still structured, experience of organization than that of the early IPO framework presented in Literature Review I (Chapter 2). This section shows how the conceptualization of teamwork within the IPO approach has developed so that it matches these experiences better. This has occurred as the researchers using this approach came to consider teams in naturalistic environments existing over long periods of time, rather than short lived or single task teams.

Time plays a role in teamwork frameworks in two senses. The first is a consideration of time scales and the second a consideration of the development and change in teams over time. It is the first which is most relevant to investigating the research questions of this dissertation.

Zaheer et al. (1999) summarizes the point that just as there are "levels of analysis" (e.g. individual, group and society) analysts ought to consider differing time scales. Analysis is possible at short time-scales such as intra-day or intra-task and at longer time scales (over months, years and organizational missions, rather than per-task). Over different timescales, different aspects of team performance will be relevant.

The developers of the IPO models had pre-figured this understanding in their work, but it didn't become fully expressed until McGrath (1991) described an emerging view of naturally occurring teams:

> *Many of the groups we meet in everyday living are not like [groups in experiments] at all. They have pasts together, and they expect to have futures. Yet they have variable membership from one occasion to another. They seldom exist in isolation; they are embedded within larger social aggregates—communities, organizations, neighborhoods, kin networks, and departments. ... Even their pursuit of production goals is often not composed of repetitive, unrelated tasks, as in successive "trials" of an ex-*

> *periment but, rather, of complex sequences of interdependent tasks that compose a larger "project." And they often have more than one such project going at the same time.*[1]

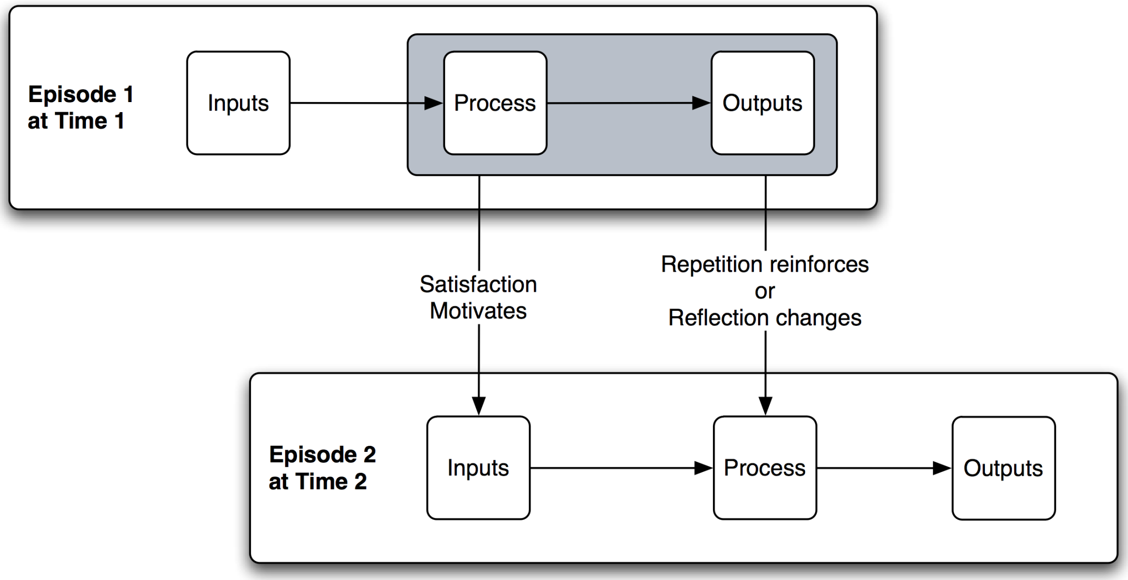
The logical combination of these two understandings of time is the introduction of an episodic conceptualization of teamwork (Kozlowski et al., 1999; Marks et al., 2001; McGrath, 1991; McGrath and Tschan, 2004). This conceptualization proposes understanding work in teams as a recurring, overlapping and developing flow of episodes, each with their own unique inputs, processes and outputs. Development is possible because earlier episodes alter both the inputs and the received practices. Episodes too, can be viewed at different timescales, with day-to-day episodes being quite short but, as the time period of analysis is expanded, together they can be seen to make up longer episodes (McGrath and Tschan, 2004). Episodes also breakdown the unitary idea of a team, since each episode might only include a small subsection of all the team members. This is true of the Tasks identified in the previous chapter; while there is some smallest unit of Task, there are also Tasks that include other tasks, such as resolving issues during the preparation of a release.

Ilgen et al. (2005, p. 519) in a recent review article argue,

> *Conceptually, team researchers have converged on a view of teams as complex, adaptive, dynamic systems. They exist in context as they perform across time. Over time and contexts, teams and their members continually cycle and recycle. They interact among themselves and with other persons in contexts. These interactions change the teams, team members, and their environments*

[1]The meaning of project and tasks in this quotation are slightly different than in their use in this dissertation. McGrath reserves the word "team" for describing the abstract organization, and uses the word "project" to describe an outcome focused organizing structure, within the life of the team, which is made up of Tasks. In contrast this dissertation does not distinguish between a project and a team, following the participant's own usage. We believe that this highlights the extent to which participant's perceive the shared outcome of the application, rather than an abstract team-ness, as the central organizing feature of FLOSS organizing.

Figure 4.1: Groups work over time in repeated episodes of Input-Process-Output. A more realistic picture would depict multiple overlapping episodes, each with effects on the other, and would acknowledge that different subsections of the team are involved in only some of these episodes.



Ilgen et al. go on to argue that the idea of IPO should be replaced with “IMOI”, standing for “Inputs, Mediators, Outputs, Inputs”. Process is replaced by Mediators to also include “emergent (psychological) states”, along with the processes of taskwork and teamwork (see Chapter 2). The Input category is repeated to emphasize the cyclical nature of teamwork over time.

McGrath and Tschan (2004) provide the most sophisticated conceptual model of teams in this tradition, aiming to draw together the IPO approach with the action theory of Tschan and von Cranach (1996) and the complex systems approach of Arrow et al. (2000) into what they call Groups as Complex Action Systems (CAST). This is an approach that emphasizes multiple layers of embeddedness, both in levels of analysis (members in groups in contexts) and in time-scales of analysis (operational processes, developmental processes and adaptive processes) which all interact to shape the life of a team.

In terms of operational processes, they argue that “group task performance is highly patterned, but not in a form reflecting a single fixed sequence of phases” (such as, say, intelligence-design-choice) but argue for a hierarchical view in which project (or goal) choice is matched with an initial choice of implementation practices (planning) which may operate unchanged if they perform, or maybe be altered if they do not perform. At the micro-level work, “unfolds in recurrent cycles (orienting-enacting-monitoring-modifying)” which are, in turn, monitored periodically at the higher/longer levels in which they are embedded.

There is no doubt that this framework is complex and dense, with the telescoping levels of embeddedness presenting a significant challenge to empirical research. Ilgen et al. (2005, p. 519) argue accordingly that, “The empirical research on teams in organizational contexts is also moving in the direction of increased complexity, but this work still has a way to go to match developments in the conceptual domain”.

Recently scholars interested in analyzing activity of online groups from their archives have also focused on this idea of episodes as a fundamental unit of analysis. Annabi et al. (2008) base their argument on Miles and Huberman (1984), rather than the literature presented above, and define episodes as “events, processes and practices that occur over time and have a beginning and an end”. They argue that these “Episodes contain the behaviors that lead to particular outcomes of interest to the researcher.” They argue that these episodes permit “systematic comparison of their occurrence, nature, antecedents and consequences”.

These conceptualizations are important to this dissertation because they suggest an empirical approach which is rooted in an understanding of work in teams as unfolding through multiple smaller groupings which overlap each other in time and involve only some members of the team. This shift in the perception of organization from macro-structures to the micro-structuring of work is mirrored in the work on coordination and dependency. In this way the empirical studies presented in Chapter 5 are a contribution to bringing empirical research closer to the increasingly sophisticated conceptualizations developed above, bringing a focus on everyday taskwork to explain teamwork processes usually studied in isolation.

4.2 Dependency and Coordination

As discussed in Literature Review I (Chapter 2) there are a myriad of ways to understand organization. The BibDesk case study, by identifying Sessions, Tasks and Discussions and suggesting links with motivation, provides an emic guide to understand the aspects of organization likely to be linked to motivation in FLOSS projects. This suggests a focus on dependencies as a crucial feature of organization. Further it is necessary to ask how the largely individual work experienced in BibDesk could be combined into a well functioning whole, which suggests a focus on coordination, which is itself closely linked with the concept of dependency. This points to a consideration of how, in practice, individuals interact and how their contribution and work relates to each other. As the literature reviewed in the following section suggests, this can be framed by asking about the manner in which individuals and their work are dependent on each other and how their work comes to be coordinated, producing a single, cohesive, shared output in the form of a software application.

Interdependency is a fundamental concept in organizational theory and has been extensively studied. This section attempts to summarize the existing literature on interdependency and it is structured in three parts:

1. What is interdependency (nature)?
2. How is interdependency related to performance (impact)?
3. What are the sources of interdependency (causes)?

What is interdependency?

Thompson (1967) is a remarkably persistent touchstone in organizational science. It describes interdependence between organizational units of a large firm, such as accounting and finance, or design and production. Thompson's description turns on a definition of interdependence that is linked to risk and contingency: dependence exists when there would be a negative outcome if an event not under the control of a single unit was not to occur in the required way. Given the potentially negative outcome, units in an interdependent system would have to readjust their activities to avoid or lessen the overall impact of other's activities.

Using this definition Thompson describes a hierarchy of interdependence. At the top is pooled interdependence which exists when two organizational units have to contribute to a common outcome. If one unit doesn't perform as needed then the outcome for the organization (and both units) might be negative, "failure of any one can threaten the whole and thus the other parts" (p. 54). Pooled interdependence doesn't require actual direct interaction; the interdependence is because the two units share an outcome (usually profits).

If two units have to directly interact for good overall performance to be achieved then Thompson describes two, more specific, types of interdependence: sequential and reciprocal. Sequential interdependence exists when the activities of one unit are required to follow the activities of another unit, if overall organizational performance is to be achieved. If the activities of the units play out over a longer time-scale, then they might need to "mutually adjust" their ongoing activities to assure overall performance. Thompson names this "reciprocal interdependence" which occurs when "each unit involved is penetrated by the other" (p. 55). Thompson makes the point that each successive type of interdependence includes the earlier. All reciprocal interdependencies include sequential interdependencies and both are types of pooled interdependency.

To put this in terms of FLOSS organization, consider the case of resolving a bug experienced by a particular user. The user cannot get the desired performance unless the bug is fixed by a developer, but the developer cannot begin fixing the bug unless the user provides a decent bug report. For the project to perform both must act in certain ways (pooled), the bug reporter must act first, followed by the developer (sequential) but in a longer term sense the bug reporter is both dependent on the developer having created the application, and is likely to provide better reporting if they monitor the developer's efforts (and vice versa) suggesting that this relationship might also be described as reciprocal.

Van de Ven et al. (1976) builds on Thompson's three types of workflow interdependence by adding "team interdependence" which they distinguish because, "in team work flow, there is no measurable temporal lapse in the flow of work between unit members, as there is in the sequential and reciprocal cases; the work is acted upon jointly and simultaneously by unit personnel at the same point in time" (p. 325). Thompson may have simply considered this to be a characteristic of work within a unit, given his concern with macro-structures in organizations, nonetheless for a study of team level organization this is an important contribution.[2]

Malone and Crowston have examined organizational dependencies in greater detail, presenting a wider ontology of relevant objects, drawing on "more precise notions" from Artificial Intelligence (Malone and Crowston, 1994; Crowston, 2003). Rather than simply examining dependencies between units, they introduce four elements: goals and tasks, actors and resources. Dependencies arise between combinations of these. Actors, specifically, are interdependent as a result of the tasks they are involved in being dependent on each other. They call the management of these

[2]It is usual in reviews of coordination to refer to Mintzberg's 5 or 6 types of coordination here, but since Mintzberg Mintzberg (1979) (and March and Simon (1958) before that) do not build their models of coordination on an explicit analysis of dependencies they are omitted.

dependencies coordination, creating a definition that has been very frequently cited (Crowston et al., 2006b).

Dependencies are examined in more detail by examining combinations of goals, tasks and resources. Goals are dependent on tasks which can, in turn, be decomposed into sub-tasks. Further detail is achieved by the observation that tasks have both pre-conditions and effects and examining permutations of relationships between tasks and resources (including actors).

The simplest dependencies in Crowston's taxonomy are between single tasks using or producing single resources (Crowston, 2003). Interdependencies between actors are likely to arise from more complex configurations such as a task using multiple resources (Shared Resource), or a task requiring one resource and producing another (Producer/Consumer).

The interfaces between tasks can have certain requirements (Malone and Crowston, 1994), such as "usability" which is the requirement that the output from a task be suitable as input for the task seeking to use the resource. Another interface requirement might be transportation or notification, where a resource required as an input needs to be in a certain location and the next task (or the actor on which the next task depends) notified of its availability.

To follow our bug fixing example, the user and the developer are interdependent only because they are linked through the goal of fixing the bug, which can be decomposed into two tasks: producing a bug-report and fixing the bug. The user, and her experience, is a crucial resource for the bug-reporting task, which produces a bug-report. The second task, fixing the bug, depends on the first because it has a bug-report as its pre-condition. Further the bug-report must be useable: it must allow the replication of the bug. The bug-fixing task depends on three resources: the bug-report, the developer and the codebase. The effect of the bug-fixing task is a patch on the source code. This patch could be seen as a pre-condition to a further task, such testing the fix, which would itself be dependent on other resources including, possibly, the original reporter as a tester.

How is interdependency related to performance?

What is shared in the conceptualizations above is that dependency matters because it creates risks for the ability of the organization to perform by meeting its overall goals. If dependencies do not “line up” as required then the tasks cannot be performed as needed and the performance of the organization will suffer. This is the essence of the “shared outcomes” condition that helps to define when an interdependency exists; if an actor’s (or an organizational unit’s) payoffs are affected by the way other’s actions integrate with their own, then there is a dependency.

Therefore the objective of analyzing dependency in the literature is to elaborate a “theory of fit” between tasks and team processes and practices (Perlow et al., 2004; Schoonhoven, 1981). An appropriate fit leads to good performance, while a inappropriate fit leads to poor performance. These theories are therefore a type of contingency theory, in which a single solution is not always the best, but different circumstances call for different solutions. In particular, practices that are called coordination are considered the primary element that should be in alignment with the requirements of dependency if organizations are to perform well.

In Thompson this alignment was understood as a one-to-one correspondence:

> *With pooled interdependence, coordination by standardization is appropriate; with sequential interdependence, coordination by plan is appropriate; and with reciprocal interdependence, coordination by mutual adjustment is called for. (p. 56)*

Malone and Crowston’s more detailed analysis argues that dependencies pose problems, such as resources being available for tasks, or that inputs for tasks are usable and in the correct place at the correct time, which if not solved will impact performance. These problems they term “coordination problems” and the techniques applied to solve them are called “coordination mechanisms” (Malone and Crowston, 1994).

Crucially, the additional features of dependencies that they describe cause different types of coordination problems and thus they are able to move beyond a one-to-one correspondence to argue that “there are many different coordination mechanisms that could be used to address the same coordination problem” (Crowston, 2003, p. 7). If the mechanism applied is successful it is able to manage the dependencies appropriately and thereby accomplish coordination. The fit between the mechanism and the problem is good and thereby organizational performance is achieved.

The literature reviewed to this point tends to assume two things: firstly that interdependencies are inherent in the task and that organizations will gravitate towards the most appropriate mechanism as a result of market forces, essentially poor performance will expose bad fit, and organizations will change, either through active management or an evolutionary process where poor performers cease to exist. As we will see below, more embedded studies of dependency and coordination argue for a different perspective in which interdependency is not inherent and is itself an organizing choice, made in conjunction with choosing an appropriate mechanism.

What are the sources of interdependency?

As discussed above, the majority of work on interdependence focuses almost exclusively on task as the source of “requirements” which must be fitted by organizational structures. So ingrained is that understanding that the definition of task is often expanded to include all aspects of the overall organizational challenge. This section first examines the literature which has questioned the structural or fixed nature of task and which opens up the possibility that organizational interdependencies occur for reasons other than an unchanging task. The review below describes this as a move from task as "pure structure" to a view of interdependence as an emergent phenomenon of practice.

Malone and Crowston (1994) decouple the analysis of structural dependancies, including those from tasks, from the analysis of processes capable of satisfying the dependancies. They retain the idea of fit, but argue that it can be achieved in many ways, albeit remaining within constraints imposed by the task.

This decoupling of task and process creates room for the observation that teams are able to successfully achieve similar tasks in different manners and raises the question: why do some teams choose to approach a task in a particular manner?

Scholars addressing this question altered the purely structural role of task by observing that interdependence in performing a task can be altered, even for identical tasks, in a number of ways (including instructions and arrangement of outcome rewards) and teams with either low or high interdependence can still perform at high levels, given alignment with features other than task.

Shea and Guzzo (1987) studied the performance of teams where a manager chose between promoting task interdependence and outcome interdependence. They found that

> *teams exercised substantial discretion in determining their patterns of interaction at work ... we regard task interdependence as malleable, especially when the group's work is not highly constrained by technology ... different patterns can prove equally effective, managers may want to give groups latitude in this area.*

Wageman (1995) intervened in the workplace to produce groups with different levels of interdependence in their work by varying outcome interdependence, i.e. making rewards contingent on the work of others. The underlying idea is that if rewards are dependent on group performance, then teams will tend to work in an interdependent way. However she found that the tasks could be successfully performed both by teams displaying interdependence and by those working individually. This result held provided that there was fit between the outcome interdependence and interdependent performance, which is to say that groups that were rewarded as teams but functioned as individuals didn't perform well (but also didn't shift to functioning in more interdependent ways).

Rico and Cohen (2006), studying online groups, argue that

> *task interdependence is a structural characteristic and at the same time, that the same task may be carried out with differing levels of interdependence . . . the technology used imposes a certain structure on the task in terms of the required level of interdependence, but it is no less true that the same task may be performed at different levels of interdependence.*

In Rico's experiment task interdependence was created by instructions, but the shape of the outcome was constrained to be identical. Despite this flexibility their conclusions retained the same 'required by the task' language, but did introduce the idea of a necessary fit with communication technologies: "the performance of [virtual teams] depends on the fit between the level of interdependence required by the task undertaken by a team and the degree of synchrony provided by available communication technologies" (p. 269).

This line of research questions whether, in fact, tasks themselves are programmed with interdependencies; it seems that teams and individuals have substantial latitude by which they can alter interdependence and still effectively perform a task, especially if other features of the production environment are allowed to vary and the groups are allowed to choose their approach.

Certainly there remains an expectation that there are limits to this flexibility. For example if a researcher artificially creates a short-term "split information" task and then requires it to be performed without any communication at all one would expect low performance. Perhaps, though, given enough time a single individual would be able to learn the needed information and thus complete the task with high performance and low interdependence. Of course almost all such laboratory tasks are set-up to have built-in time limits. This is an assumption that makes sense if one is focused on organizational environments where efficient use of resources is paramount, but it is an assumption nonetheless.

Finally there are scholars examining coordination by studying the details of actual practices in naturally occurring teams over time. Wageman and colleagues (Wageman, 1995; Wageman and Gordon, 2005) provide a useful bridge from the "structural but interpreted" work described above and the practice investigations discussed below. They separate structural from emergent causes of interdependence. They outline four sources of "*structural* task interdependence" (p. 688, their emphasis):

1. How the task is defined to the group,
2. Rules that managers establish about process (instructions and expectations),
3. Physical technology of the task (production line), and
4. Managerial allocation of resources necessary for the work.

They argue, though, that these factors can be dominated, without loss of performance, by group characteristics which make interdependence an "*emergent* property in social systems" (p. 688). In particular they examine the effect of the group's values on the manner in which they arrange their work. Groups with shared egalitarian values arranged their work with higher interdependencies and cooperation, while groups sharing meritocratic values arranged their work with lower interdependencies and cooperation. Both were successful; and significantly more so than groups without shared distributive values. In earlier work Wageman (1995, p. 177) also argued that congruence between dependencies and team structure may only lead to good performance, "as long as the work itself is motivationally well designed, the groups well composed, and the individual members competent."

They argue that the dominance of "socially emergent" interdependence over "structural" interdependence can occur in teams where "the structural features of the task typically are either ambiguous or are left unspecified" and cite knowledge work, consulting and new product development as examples.

This work opens up the possibility that the necessary fit between task and appropriate interdependency and coordination mechanisms is related to task, but only in so much that some tasks have firm requirements, while others are flexible enough to be able to be performed in different modes, allowing teams to choose their levels of interdependence without performance penalties. It seems clear that in a volunteer environment undertaking software development there are very few "structural" sources of interdependence, leaving interdependence to "emerge" in a manner which best fits these other contextual constraints.

This work shares the understanding that interdependence in task performance emerges through an episodic process of structuration, where initial tendencies (structural, individual, contextual) are strengthened by repeated successful practice and "the patterned behavior of groups becomes normative and process solidifies into structure." (Wageman, 1995).

Researchers employing a practice lens to explore in detail the work of teams have also observed this pattern of structuration, a co-formation of reinforcing practices of interdependence. Perlow et al. (2004) examined software engineering teams in three international locations pursuing similar tasks. They observed quite different patterns of interdependence (Manager centered, Expertise centered and Team centered) but similar levels of performance. They argue that, in each environment, practices have emerged that fit each other as well as the (highly flexible) "nature of the task", the external environment (e.g. national cultures) and individuals, "the types of people who are hired". They turn to structuration, with its mutual reinforcement to explain how this fit co-evolves. They argue that action to improve performance requires "not just understanding the patterns of interaction themselves but also the organizational, institutional, and cultural contexts that enable and constrain them" (p 534).

Faraj and Xiao (2006) studied work in a medical trauma unit, which they characterize as "contextualized and non-routine" but also are "high velocity" and must "avoid errors". They identify the importance of improvised ways of coordinating, based on relationships, but also argue that "formal modes of coordination do not melt away in favour of more improvised ways of coordinating". While there are structures that are important, the group is largely responding to "the situation" at hand and actively re-planning their work, continually choosing patterns which may or may not be interdependent in execution.

The point is clear: as researchers have studied groups undertaking less structured and repeated tasks in organizations that persist for longer periods of time the "structural requirements" of tasks become relatively less important in predicting the overall level of interdependence likely to be seen in the group's activities. Rather the literature highlights an evolving fit between the preferences of the individuals involved, affordances of available technologies and patterns of interdependence.

To those two determinants ought to be added the circumstances of the individuals involved. One of the key aspects of the resource context of FLOSS teams is that they are emergent and voluntary, in the sense that their participants are not assigned, but must be attracted. They are not rewarded for their contribution with external rewards provided by an organization, such as money. Rather, as discussed above in the review of literature on motivation in FLOSS teams, they derive their rewards, whether intrinsic or extrinsic, from the work itself. Furthermore the project's utilize a technological infrastructure that has been intentionally developed to support work in these circumstances. The need to synchronize these factors ought to shape the type of interdependence and thus organization to be found in FLOSS taskwork.

Interdependency in Software Engineering

Interdependency also plays an important role in a separate tradition of literature from Software Engineering which considers the role of modularity in a code base. Modularity is a property of software code whereby different sections of the code are low in their interactions or "interpenetration", usually called "coupling and cohesion". For example highly modular code does not share 'global' data structures and draws on functions only through a small well-defined set of high level function calls, known as an interface.

It is argued that effective parallel development is achieved through such modular code, and that it leads to higher quality code with fewer defects (Baldwin and Clark, 2001; Langlois, 2002; von Hippel, 1990; Parnas et al., 1981). This has recently been examined and argued as a factor for successful open source software as well (Conley, 2008). In short, a good design ensures that people can work relatively alone, since they are only optimizing and improving their own sections.

The received wisdom in Software Engineering is that the structure of an organization determines the structure of the software which it produces. This is known as Conway's law, "organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations" (Conway, 1968) with a common expression being "If you have four groups working on a compiler, you'll get a 4-pass compiler"[3]. (Notice that the "law" specifically refers to *designs*). Recently more detailed empirical studies in Software Engineering have begun to critique these understandings, drawing on more nuanced literature from organizational science (e.g. Herbsleb and Grinter, 1999).

The question of the role of modularity in the codebase in relation to the sorts of independent work patterns observed in BibDesk is a valid one and will be returned

[3]http://en.wikipedia.org/wiki/Conway%27s_Law

to in the Discussion (Chapter 8), along with questions of the impact of motivations and resource context.

4.3 Literature Review II: Conclusions

The literature review in this chapter helps to focus the understandings generated in the BibDesk case study and suggests an empirical focus on episodic, repeated taskwork as the type of answer to RQ1 which is most likely to play a role in the answer to RQ2. The dissertation now turns to describe a systematic replication of the central findings of the BibDesk case study using the conceptualization of work presented in this literature review.

Chapter 5

Replication: Fire & Gaim Archive Study

5.1 Introduction

This chapter presents a systematic study of work practices in two FLOSS projects. It was designed to replicate key findings of the BibDesk participant observation and provides additional illustrative data for the theory developed in Chapter 7. The key findings of the BibDesk case study were that work is largely individual and often productively deferred. The study reported in this chapter replicates and focuses these findings through a systematic classification of tasks extracted from archival evidence. The analysis is based on a combination of emic understanding gained during the BibDesk study, the literature reviewed in Chapter 4 and immersive reading and iterative organization of a full set of project archives. A set of concepts emerges from this process and are used to build a classification of types of collaboration in the projects studied.

Scope and specific research questions

While the BibDesk case study considered all types of activity in the project, this study focuses on only work which leads to changes in the shared outputs of the projects. For the very large part that is work that leads to changes in the codebase that produces the application, but it also includes changes to documentation on how to use the software. This choice is a trade off; it allows easier comparison of like to like in the systematic results, but clearly reduces coverage of the project's activities. For example it loses activities such as user support, design discussions between developers that do not result in implemented changes and other assorted non-production activities. Nonetheless it focuses on activities that are literally vital to the projects; at their core FLOSS projects are different from other online communities because they are production communities and without developing code they quickly die.

The scope of this study is limited, therefore, to the patterns identified in the previous chapter as Tasks, as opposed to Discussions or Sessions, and then only to a particular—albeit it core—type of Task. This chapter therefore has more specific research questions than the dissertation overall:

1. What patterns of collaboration are found in task-level production activity in Fire & Gaim? Does individual taskwork dominate?
2. What evidence is there for deferral as a strategy?

Method

The method employed for this study was both qualitative and systematic, taking particular aspects of my experience in BibDesk and re-deploying them in a focused study of the comprehensive archives of the projects studied in this chapter.

It is a feature of FLOSS work, as reported in Chapter 3, that it is almost entirely experienced through documents. In the offline world archives can be copious but are at best partial records of work because much occurs face to face and creates no record. In FLOSS projects, by contrast, because documents are the *medium* of work, archival documents provide a far more comprehensive record. Even years later, such archives give the qualitative researcher, skilled in the interpretative milieu through his own participant experience, access to the project's work in a manner very similar to that which the project members had.

This access, of course, is limited in the ways described in Chapter 3 and care must always be taken to recognize the impact of the mechanisms of archiving, (what Hill (1993) calls "sedimentation practices" in historical archival research) as well as the limitations resulting from archives only capturing the public aspect of the full FLOSS working environment. For example it would go too far to claim that reading the archives alone, without interviews or other probing, gives insight into the motivations of individual participants (even real-time participation arguably doesn't do this). One cannot engage in real-time probes, either through email or interviews, to assist one's interpretations. The claim is not that this archival study is an ethnography, but more simply that it an uncommonly rich qualitative archival study.

In any case, happily the challenge of this chapter is far simpler than that of deeper ethnography. The challenge is to recover the set of tasks which the experience in BibDesk held to cohere and organize the work of FLOSS participants. In this task two key issues noted in Chapter 3 arise. The first is the fragmentation of evidence about work into multiple separate archives (different email lists, trackers etc). The second is the compressed and flattened experience of time. Rather than the FLOSS project winding in and out of one's "real life" and thereby periodically re-connecting and recalling the issues "on deck", an analyst reading archives experiences time as far more linear and regular. This means that particular attention must be paid to the time-stamps and, in particular, the gaps between working Sessions and the Bursts they create (see Chapter 3).

These two challenges require that systematic qualitative work to make sense of archives must be exhaustive (covering all archives) and iterative; initial organizing categories must be re-considered as more evidence comes to light, and the interpretation of already read evidence can be altered by evidence read later. An analyst is assisted in this because the narrative and temporal coherence of the tasks is a constant pull towards discovering the underlying and generative task structure. This may seem to put the cart before the horse, but the task is analogous to unpacking a mobile from a box; gentle shakes and gravity allow the existing structure to assert itself over the original jumble.[1]

In this way a careful, systematic, iterative and exhaustive reading and organization of the project archives by an analyst attuned to the working practices of FLOSS projects has the potential to answer the relatively simple research questions of this chapter. This method, of course, comes with limitations which are dealt with in detail below together with their potential impacts on the results and suggestions for improving their robustness in future research.

Case Selection

The two projects selected for this study are Fire and Gaim. Their selection is justified because they provide good cases to replicate the findings of BibDesk. This replication logic—rather than a logic of sampling or coverage—is recommended for case-based research (Yin, 1994). For this dissertation, based as it is on a socio-technical argument, the key risk in the BibDesk study is that the observed practices were either purely idiosyncratic to BibDesk, or are purely social, as opposed to socio-technical in nature.

[1]A mobile is the type of sculpture associated with Calder and Duchamp of many parts connected by wires intended to be hung from the ceiling. Similar structures are often hung above baby's crib, their balanced, gentle motion entertaining the child.

Fire and Gaim were first encountered by the author again through personal need. I was beginning to use IM more and had recently moved countries to begin my PhD. A feature of IM usage is that different protocols/networks tend to be dominant in particular social and geographical groups, most likely due to strong network effects. To interact which all of my friends I had to be connected to multiple networks, which meant having three or more IM applications open on my computer. This is inconvenient and confusing, so I looked for alternatives and discovered Fire, since I was using Mac OS X at the time. I discovered the application during its most the successful phase, in 2003. I was never a developer for Fire, but I did follow the user and developer lists while I used the software. Gaim was later selected as a useful comparison project and this comparison has been used in other content analytic research which, through continual reading of the archives, provided a broad-stroke understanding of the projects (e.g. Scialdone et al., 2008). This research was conducted at the same time as the participant observation in BibDesk and, together with other FLOSS projects I was involved in, provided informal comparisons.

Since the overall argument is one of replication it is sensible to choose cases which are similar enough to BibDesk on the features thought to matter for independence, as discussed in Chapter 4: Resource context, Task and Infrastructure. Fire and Gaim are both community founded projects which, at the time of the study, had no institutional structure and the work of which was performed entirely by volunteers. They have no foundations or other formal existence. The demographics of the participants are not known other than as they are revealed through the communications, which suggests that many participants were students—as with BibDesk—although on the whole they seem younger, including what appear to be high-school students. That said, other contributors are clearly older and do not self-identify as students. The participants are distributed globally, although they are focused in North America and Europe.

The projects are as close as possible to each other, as well as BibDesk, in terms of technical infrastructure. They are both hosted almost exclusively on Sourceforge. They both used CVS at the time of the study (and just like BibDesk switched to Sourceforge's SVN after it was offered). They both use Trackers of basically the same types (bugs, enhancement requests etc), although Gaim uses more to track plugins and patches. They both have mailing lists roughly divided into development and user focused venues, although Gaim employs a Sourceforge-hosted user Forum rather than a users mailing list. Both have had sundry non-sourceforge venues, but in the period selected for the study had their collective life focused in Sourceforge hosted venues.

The overall tasks of Fire and Gaim are almost identical: they both build a multi-protocol instant messaging client. Such clients provide a unified interface to the many competing standards and networks for instant messaging: Yahoo, MSN, AOL, Jabber and IRC. They are both user interface heavy, end-user software that seeks to solve a personal productivity challenge in that they rescue their users from having to have multiple clients, one for each network, open and to separately manage lists of their contacts in each. The domain of the software, therefore, is not identical to BibDesk, but it is very close; both are personal productivity tools intended for everyday use on personal computers. This is in sharp contrast to software intended for organizational, development or periodic use (such as an ERP system, a programming library, or a special event management system, like Open Conference System). Again, as with BibDesk, neither of the projects spawned a commercial product.

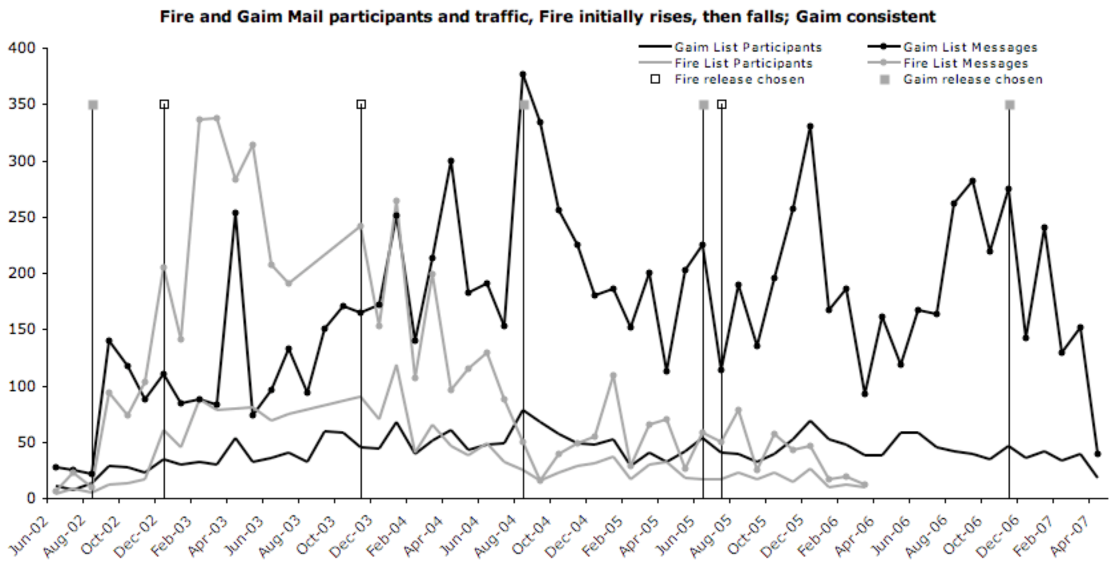
Furthermore, as will become clear below, the projects both rely, as did BibDesk, on underlying libraries built in a surrounding project ecosystem. They therefore have the challenge of bringing these together in a UI as their primary programming task. The complexity of that integration task is probably higher for Fire and Gaim than for BibDesk, since they must abstract the different and overlapping functions of the various IM protocols. Gaim, perhaps, is slightly different in that that project also took on the task of building and maintaining an underlying library for the AOL instant messenger network: libfaim (eventually called libprpl, pronounced "lib-purple"). Both projects draw on a UI framework which provided an interface to underlying operating system and basic networking functions, GTK+ for Gaim and Cocoa for Fire (the same library that BibDesk used).

The selection of these two projects is not without tradeoffs. By focusing on similarity, rather than difference, the selection lowers the ability of the research to generalize to other kinds of FLOSS projects. For example all three of the studied projects are around the same size, in terms of developer numbers. This, therefore, may hide interesting differences on this score; perhaps very small teams are more likely to work together, or perhaps very large teams are more likely to break up into smaller teams, where more collaboration may occur. There is also a trade off in terms of domain. Each project concerns an application which is primarily designed to be used by individuals and requires little knowledge of the organizational systems in which they are embedded. This is in contrast to, for example, Enterprise Resource Planning systems where the knowledge to implement an organization's processes in software are likely to be scattered throughout organizations, perhaps leading to a greater need to work together on separate tasks, or at least to plan them in sequence and articulation. Nonetheless, if the results from BibDesk do not hold even in similar projects then there is little chance they will hold more broadly, so it is with similar projects that the work of replication must begin.

Period Selection

In terms of project success, Fire and Gaim exhibit trajectories with both similarities and differences. Both projects begin small, limited to a single minority platform (Fire to Mac OS X, Gaim to Linux). Initially both are extremely successful, relative to most FLOSS projects, and rise to be the number one multi-protocol client on

Figure 5.1: Fire and Gaim have comparable participation through early 2004; after this Fire falls away, while Gaim moves from strength to strength



their respective platform for a period. Both were considered innovative and exciting FLOSS projects. They have similar numbers of registered developers, in the top 2% of Sourceforge projects at between 10-15. Overall Gaim always has a large user base, as proxied by downloads and has more activity across its various community project venues, although levels of CVS activity are similar.

After around 1.5 to 2 years, during 2004, their trajectories diverge, with Gaim moving onward to further and larger scale success and Fire falling away, eventually declaring themselves finished, although leaving all the project infrastructure in place in case others wish to take up the mantle. The remaining Fire developers decamped to another Mac OS X multi-protocol instant messaging client, AMSN. These effects are shown in Figure 5.1, Figure 5.2 and Figure 5.3.

Figure 5.2: Fire and Gaim have comparable numbers of registered developers through May 05, after which Gaim climbs rapidly

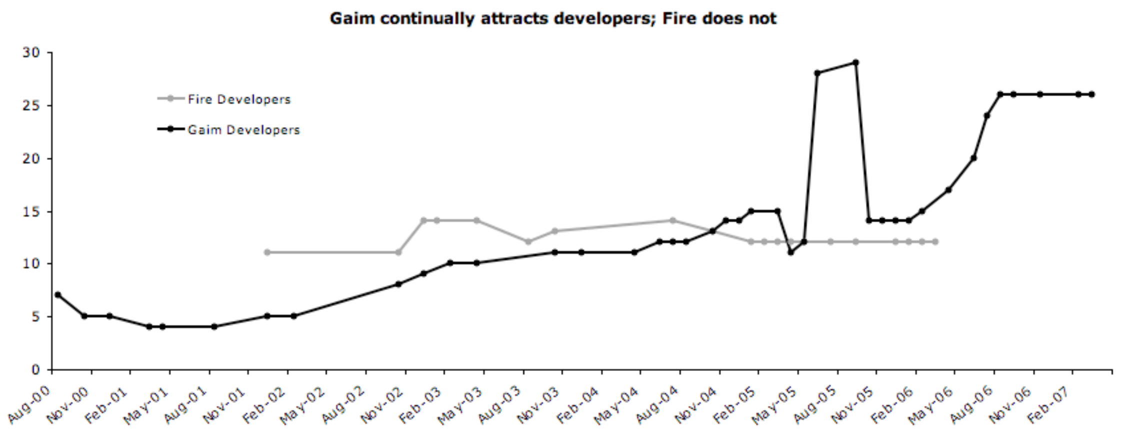
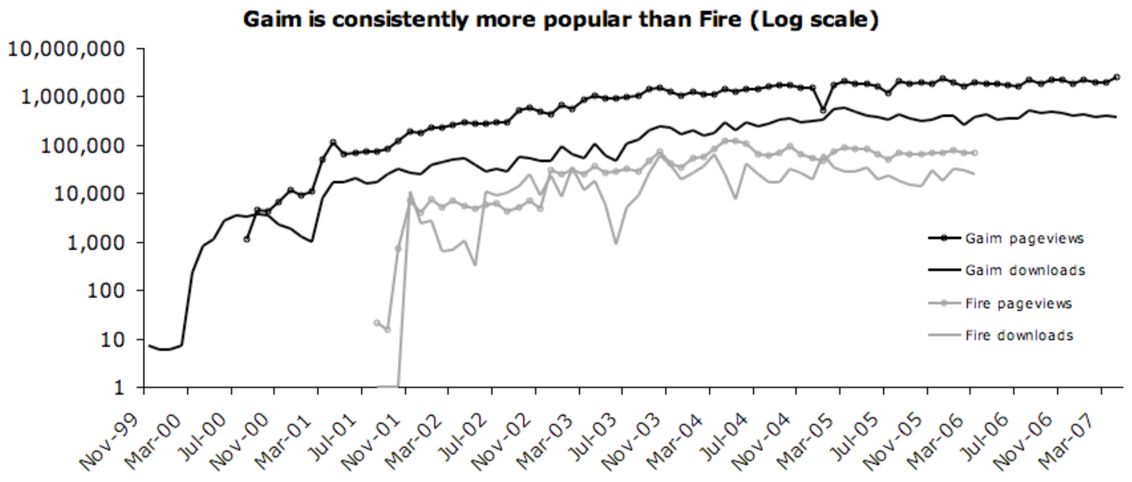


Figure 5.3: Gaim always had more users and interest than Fire, although the installed base of their initial platforms (Mac OS X and Linux) was comparable.



The similarities and differences can be clearly seen in Figure 5.1, Figure 5.2 and Figure 5.3. For this reason this study chose to concentrate on a period for which the projects are most broadly comparable, especially in terms of community participants and registered developer numbers. The periods were also selected to be comparable in terms of length and size of work undertaken in the period. This criteria resulted in selecting the periods Fire 0.32.a and Gaim 0.59.2. The inter-release period for Fire was 56 days (16 October 2002–11 December 2002), while for Gaim it was 61 days (24 Jun 2002–24 Aug 2002). The nearby calendar periods help ensure that the projects are facing similar challenges during the period, since part of the work of the team is responding to changes in the underlying protocols and to the availability of libraries and other code in the open source instant messaging ecosystem.

5.2 Data

The raw data for the study was collected to be as comprehensive as possible. Data was obtained from four sources: the OSSmole project (Mailing lists) (Howison et al., 2006), the Sourceforge Research Archive at Notre Dame (Trackers, Forums and Releases) (Greg Madey (ed), 2007; Gao et al., 2007), XML export from the Sourceforge SVN for each project, and by using the ViewVC browser for details of changed files and the changes themselves. The entire archives for the projects were collected—not just the single chosen period—which allows these periods to be placed into context.

The data collection was substantial, but falls short of fully comprehensive, in part for reasons of public/private cycles discussed on page 54 (Chapter 3). These shortcomings must be considered in interpreting the results. For example there is evidence that both teams used IRC (Internet Relay Chat) as well as IM (Instant Messaging) between the developers and as an additional general project venue. For example for a short period of time (outside the time considered in this study) Fire held synchronous meetings over IRC, called “Fireside Chats”. This is unsurprising given that the projects were developing IRC and IM clients. It was not possible to collect this data because neither project archived it; indeed it is often considered inappropriate to archive these transient conversations (although not unheard of). However there is little evidence that the interactions in these venues played strong roles in the conduct of task work, for example IM or IRC discussions are not referred to in SVN messages or mailing list discussions. Indeed, given the difficulties the projects had with the efforts of the IM networks to block them out, it is perhaps not too surprising that they sought to communicate in other media. Nonetheless this remains a limitation of the study.

Each archival record has its own format and records aspects of the work in subtly different ways. For example participants are identified in different ways (email vs username) and time stamps refer to different time zones (UTC vs EST, since Sourceforge is based in Virginia). The data sources, even when obtaining data from OSSmole and SRDA came in different formats. For example there was direct SQL access to OSSmole data but only web-form SQL access to SRDA. Since both projects had moved their source code repository to SVN, the CVS data was available only after it had been imported to SVN, which introduces some artifacts. SVN/CVS data was obtained through an XML export from Sourceforge’s servers.

In order to organize the data for analysis a data schema was designed which allowed the raw data to be represented in sufficient detail so as not to lose the differences between venues, but at the same time to highlight the conceptual similarities (see below) that the various archives sometimes hid due to their individual data representations. This was accomplished using RDF (Resource Description Format), together with some features of OWL (a semantic web classifying language). This data integration was quite a challenge and is described as needed below, as well as in (Howison, 2008). It is relevant that the identifiers for the individual data elements in RDF are

URIs, and I was therefore able to use the actual URLs (*http://sourceforge.net/...*) as the identifiers in the database, seen in footnotes throughout this dissertation.

Over 158,000 documents were collected for the full lifetime of the two projects. The overall lifetime media usage of Fire and Gaim is shown in Figure 5.4. Of particular note is the dominance of Trackers and Forums over Mailing Lists in the Gaim project as well as the similar absolute levels of CVS activity, particularly in the first half of Fire's life time (pre mid-2004) (although this is obscured somewhat by the 5x difference in activity in Gaim than Fire).

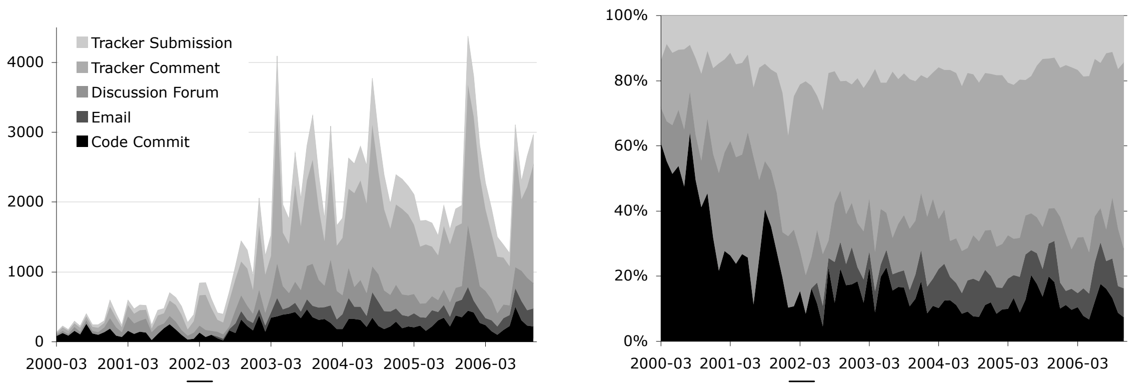
The documents for each project were extracted from the database by Java scripts, outputting each Venue in a separate text file, indexed by their URI, with the relevant messages grouped by Threads (or Tracker Items) and then chronological order. The files for each project were arranged into a folder and that folder opened in an editor designed for programming which allows easy browsing and searching through a hierarchy of text documents. This arrangement allowed flexible traversal of the document set, without the need to write an interface to the database. Further, because the data were represented by actual URLs, it was almost always possible to return to the relevant web page on Sourceforge and check for any additional useful details or processing errors.

Discovering and refining sense-making concepts

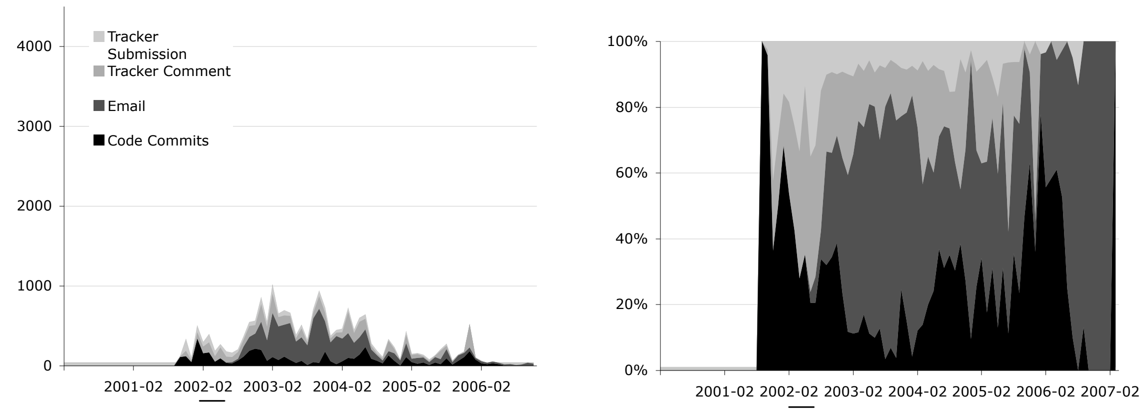
The research in this chapter did not begin in a situation of *tabla rasa*, rather the research began with concepts arising from a combination of the participant observation experience of BibDesk together with the literature discussed in Chapter 4. These understandings, however, were refined as I began to made sense of the data collection and became more familiar with the working practices of Fire and Gaim.

Figure 5.4: All Communications over time for Fire and Gaim. Note the relatively low relative use of email in Gaim, due to heavy use of Trackers. In both projects code commits are relatively stable over the life of the project, while others rise, causing a decline in relative terms, while the projects are successful. The bars under the time axis approximately show the selected periods.

Gaim: All communication archives (Note relative heavy use of Trackers)



Fire: All communication archives (Note later start and no Discussion Forum)



The data collection strategy described above brought all the data together and allowed me to re-experience it in the order it happened, gradually sorting out ways to make more systematic sense of the data. The work began in an open mode, similar to that described for BibDesk on page 59, examining archival records and seeking the narrative sense contained in them. This immersion was contemporaneous with the literature review described in Chapter 4.

There were four over-arching sense-making tasks undertaken as part of this qualitative analysis. The first was to organize the diverse archives in a way that they could all be presented in temporal order, overcoming the 'silos' into which the different records came to be stored. The second task was more interpretative, it was to recover the individual tasks which provide narrative cohesion to the work. The third task was to understand the different types of contribution to the outcomes of the tasks. Finally patterns in these contributions were used to generate a classification that speaks directly to the research question of this chapter. The analysis undertaken was iterative, which means that interpretative tensions at a later stage motivated changes in the earlier organizing constructs. This will be demonstrated by a number of examples.

Organizing the archives

The first task was to take the archival records and arrange them in such a way that they could be organized into temporal order. The chosen method was to automatically parse surface features and to store the records in a combined database, so that they could be output in temporal order. However since different archival records have different "sedimentation practices" (Hill, 1993) which might affect later interpretation, it was thought important to retain all the metadata.

The information modeling process was undertaken at first by using modeling tools from the world of relational databases, chiefly by drawing Entity-Relationship diagrams, and then writing code to organize small sample datasets.

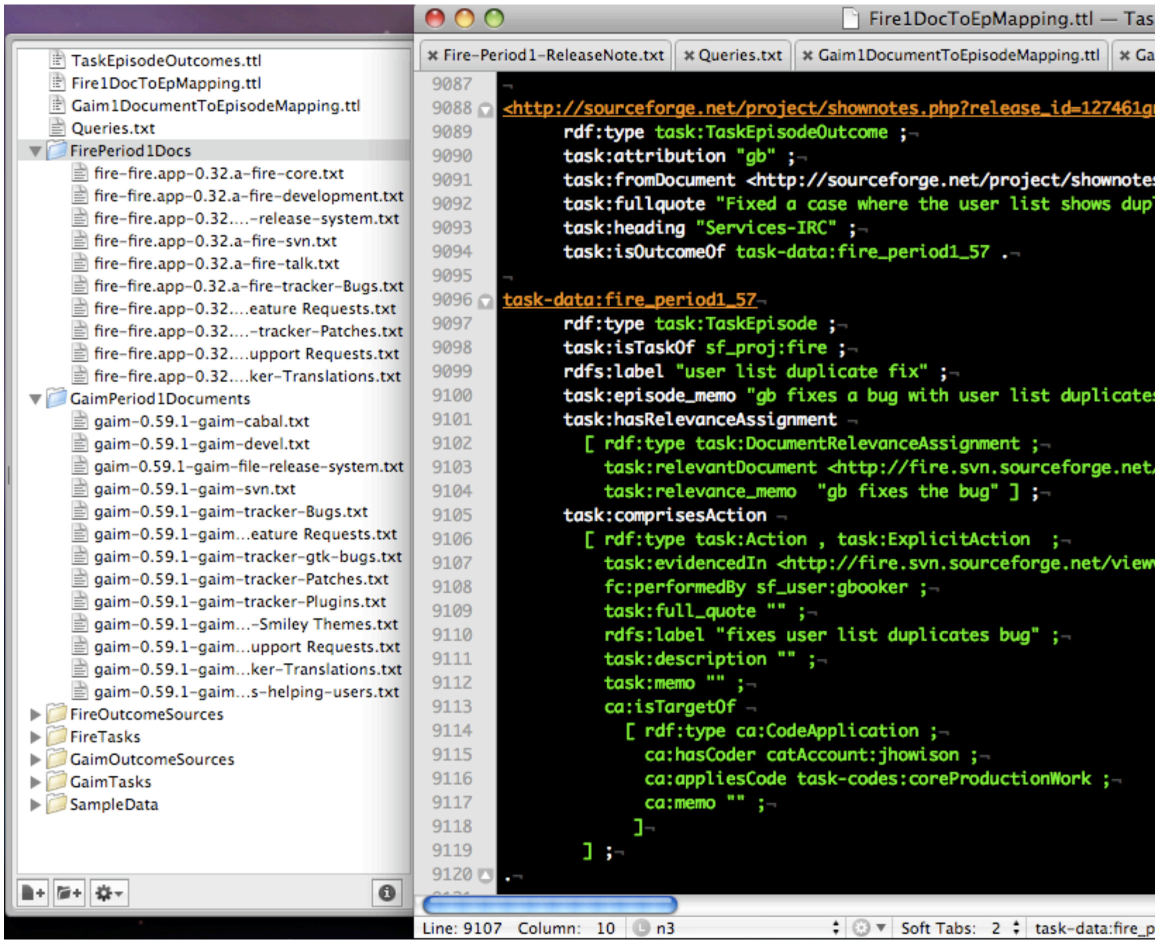
The challenge is identifying the right level of abstraction to preserve the differences, while working with the similarities between records. Another challenge was preserving the original identities of the archival records, in case I later needed to examine the document in context.

Modeling this data using traditional relational database schemas proved to be quite difficult. This is primarily because the central similarities derive from viewing them in a class hierarchy. This is to say that, for example, an email message and a forum message, as well as their attributes, are linked by moving to a higher level of abstraction rather than relational linkages. I do not mean to argue that the data could not be represented in a relational database, simply that the process of modeling the data in this way was not contributing to an intuitive understanding of the data.

For this reason I turned to modeling techniques most associated with semantic web research, known as RDF (Resource Description Format) and OWL (Web Ontology Language). These techniques naturally encourage a class hierarchy view of the data, as well as using URLs for identifying objects, which met the additional criteria of maintaining a strong link to the original data sources. Finally the schema of qualitative research evolves along with an analyst's understandings, and the RDF format allows one to generate new Objects and Properties simply, without changing existing work. I worked not with the XML serialization of RDF but with a serialization known as Turtle, which is a plain text format allowing me to manage data entry in a simple text editor, see Figure 5.5. I was also able to use convenient templates ('tab triggers') to enter repetitive portions quickly.

Working first with Email and Forum messages, then with Trackers, then with Release Notes and then with SVN log messages, I developed a vocabulary of classes

Figure 5.5: A screenshot of the interpretative apparatus. This is a plain text editor. On the left is a hierarchy of files, including the Documents from the archive, organized by archive type and date. The middle shows a document written in Turtle format, showing my understanding of a small Task. Note the various memos.



```
TaskEpisodeOutcomes.ttl
Fire1DocToEpMapping.ttl
Gaim1DocumentToEpisodeMapping.ttl
Queries.txt
▼ FirePeriod1Docs
    fire-fire.app-0.32.a-fire-core.txt
    fire-fire.app-0.32.a-fire-development.txt
    fire-fire.app-0.32....-release-system.txt
    fire-fire.app-0.32.a-fire-svn.txt
    fire-fire.app-0.32.a-fire-talk.txt
    fire-fire.app-0.32.a-fire-tracker-Bugs.txt
    fire-fire.app-0.32....eature Requests.txt
    fire-fire.app-0.32....-tracker-Patches.txt
    fire-fire.app-0.32....upport Requests.txt
    fire-fire.app-0.32....ker-Translations.txt
▼ GaimPeriod1Documents
    gaim-0.59.1-gaim-cabal.txt
    gaim-0.59.1-gaim-devel.txt
    gaim-0.59.1-gaim-file-release-system.txt
    gaim-0.59.1-gaim-svn.txt
    gaim-0.59.1-gaim-tracker-Bugs.txt
    gaim-0.59.1-gaim...eature Requests.txt
    gaim-0.59.1-gaim-tracker-gtk-bugs.txt
    gaim-0.59.1-gaim-tracker-Patches.txt
    gaim-0.59.1-gaim-tracker-Plugins.txt
    gaim-0.59.1-gaim...-Smiley Themes.txt
    gaim-0.59.1-gaim...upport Requests.txt
    gaim-0.59.1-gaim...ker-Translations.txt
    gaim-0.59.1-gaim...s-helping-users.txt
▶ FireOutcomeSources
▶ FireTasks
▶ GaimOutcomeSources
▶ GaimTasks
▶ SampleData
```

Fire1DocToEpMapping.ttl — Tas

× Fire-Period1-ReleaseNote.txt | × Queries.txt | × Gaim1DocumentToEpisodeMapping.ttl | × Ga

```

<http://sourceforge.net/project/shownotes.php?release_id=127461g
      rdf:type task:TaskEpisodeOutcome ;
      task:attribution "gb" ;
      task:fromDocument <http://sourceforge.net/project/shownote
      task:fullquote "Fixed a case where the user list shows dup
      task:heading "Services-IRC" ;
      task:isOutcomeOf task-data:fire_period1_57 .

task-data:fire_period1_57
      rdf:type task:TaskEpisode ;
      task:isTaskOf sf_proj:fire ;
      rdfs:label "user list duplicate fix" ;
      task:episode_memo "gb fixes a bug with user list duplicate
      task:hasRelevanceAssignment
        [ rdf:type task:DocumentRelevanceAssignment ;
          task:relevantDocument <http://fire.svn.sourceforge.net/
          task:relevance_memo  "gb fixes the bug" ] ;
      task:comprisesAction
        [ rdf:type task:Action , task:ExplicitAction  ;
          task:evidencedIn <http://fire.svn.sourceforge.net/view
          fc:performedBy sf_user:gbooker ;
          task:full_quote "" ;
          rdfs:label "fixes user list duplicates bug" ;
          task:description "" ;
          task:memo "" ;
          ca:isTargetOf
            [ rdf:type ca:CodeApplication ;
              ca:hasCoder catAccount:jhowison ;
              ca:appliesCode task-codes:coreProductionWork ;
              ca:memo "" ;
             ]
        ] ;
.
```

Line: 9107 Column: 10 n3 Soft Tabs: 2 task-data:fire_p and properties linking them which encompassed the whole data set, while maintaining the individual specificity of each archived document. The highest level constructs are shown in Table 5.1: Events, Documents, Participants and Identifiers.

Table 5.1: Archive Concepts

Concept	Definition *Example*
Event	An Event, occurring at a particular time, causes Documents to be archived. *–Sending an email, releasing a version*
Document	Archived content *–Email content, Tracker comment content, Release note*
Participant	A distinct individual involved with the project *–James Howison (the person), Sean Egan (the person)*
Identifier	A string identifying a Participant *–James Howison (the name), james@howison.name (an email address)*

The archival records are all linked at the highest level because they are generated by some Event. This Event, such as sending an email or submitting a Tracker Comment, or even releasing a piece of software, causes a record in the archives. It occurs at a particular time. The Events I was particularly interested in were Events which generated textual evidence, and this evidence I called a Document. Events are associated with particular identifiers, like Sourceforge user names or email addresses which refer to the real people participating in the project. I therefore created a new class of entity, a Participant, which can be associated with one or more Identifiers. Each concept has a specific sub-class for maintaining the more specific aspects of a record, thus an EmailEvent is a sub-type of Event, it is associated with an EmailAddressIdentifier and a RealNameIdentifier and its content is a EmailMessageDocument. By inspecting each type of archive I was able to normalize the times of the Events and therefore output the data in an absolute temporal order.

An example of the iterative sense-making undertaken even at this stage is the separation of Event from Document. This is necessary because some Events generate more than one Document, particularly the Release Notes. To understand the archival record one must understand the process as enacted by the participants. When preparing a release a developer (the Participant) creates an updated binary of the application and fills out a form at Sourceforge. This form has three parts: the application binary for upload, a set of Release Notes and a Change Log. For some projects the Release Note and the Change Log are identical, but others use the Change Log to list changes, and the Release Note as a more discursive announcement. When the developer presses upload (the Event) this creates three separate Documents. In the archival record the Participant in this case is identified by their Sourceforge User ID, the Identifier.

Throughout this process I was immersing myself in the data, reading the content of Documents, returning to the original records and using my FLOSS experience to place myself in the shoes of the participants and understand how their participation lead to these archives.

Recognizing Tasks

The central cohering concept identified in the BibDesk case study is that of the Task. Participant observation found this to be an interpretative concept that enabled participants to make sense of the flow of activity in the project. It is also crucial for the research question of this chapter, since they provide the unit of analysis inside which one is looking for collaboration and/or individual work. The interpretative work described in this section was informed both by the participation in BibDesk and the literature presented in Chapter 4, which understand the work of a team to be multiple, overlapping and embedded (McGrath, 1991; McGrath and Tschan, 2004).

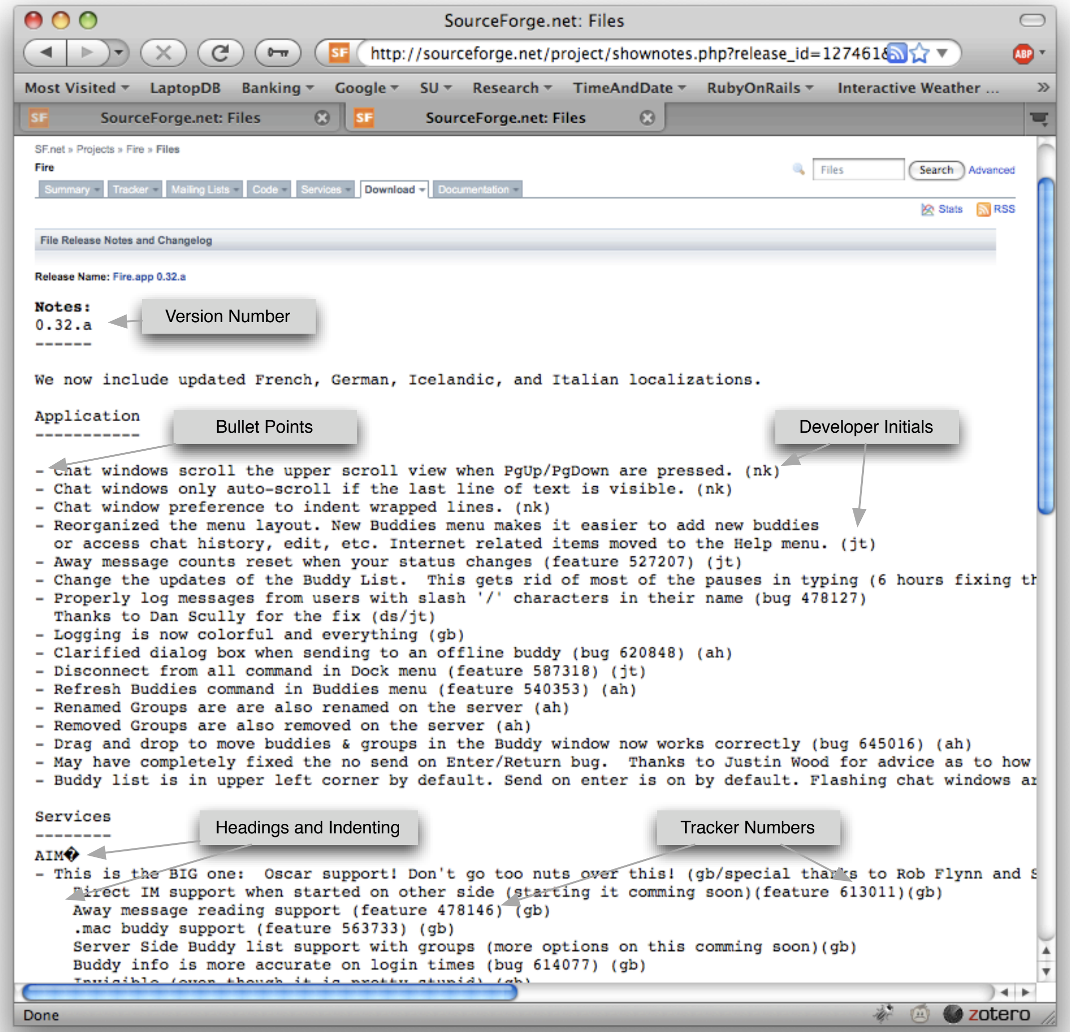
Thus an analyst attempting to recognize Tasks faces difficult decisions about the level of inter-relatedness and embedding. These are often associated with the time scale of analysis (Zaheer et al., 1999). For example it is quite natural and useful to consider each release of the software a Task for the group, and all of the contributions towards that release part of that "Release Task". Similarly when the participants are working towards a release it is reasonable to consider the last few days a push to release as a single Task that has components at a lower level as bugs are fixed or new features polished.

For the research question of this study, however, it was more useful to resolve Task at a fairly low level, but not to consider every single change to CVS to be its own Task, because clearly some are related to the same intended outcome. Finding a consistent, yet meaningful level at which to draw the Task boundaries was crucial to this study.

If the participants truly are using Tasks to provide coherence to their work, then there should be evidence of that produced by the participants themselves. There are indeed two good sources of this information. The first is the Release Notes, or Change Log, published with a new release. These documents communicate to the users what has changed during an inter-release period. They are prepared by whomever is uploading the release, since they have to be pasted into a web form on the Sourceforge site. They are the first source of outcomes for recognizing Tasks, providing outcomes for 65 of the 109 tasks ($\approx$ 60%).

However this process, coming as it does only at the end of the release period often summarizes too much, pulling together similar events "other UI tweaks"; after all it is not an accounting of the project's work, but a communication to the application's users. Happily both Fire and Gaim also use a README document, which develops through the inter-release period and forms the basis for the release manager to pull together the release notes. The README is shown in Figure 5.7. It is often the case that developers, having gotten a feature to work or having fixed a bug will alter the README file (through a CVS check-in) to communicate to other developers that

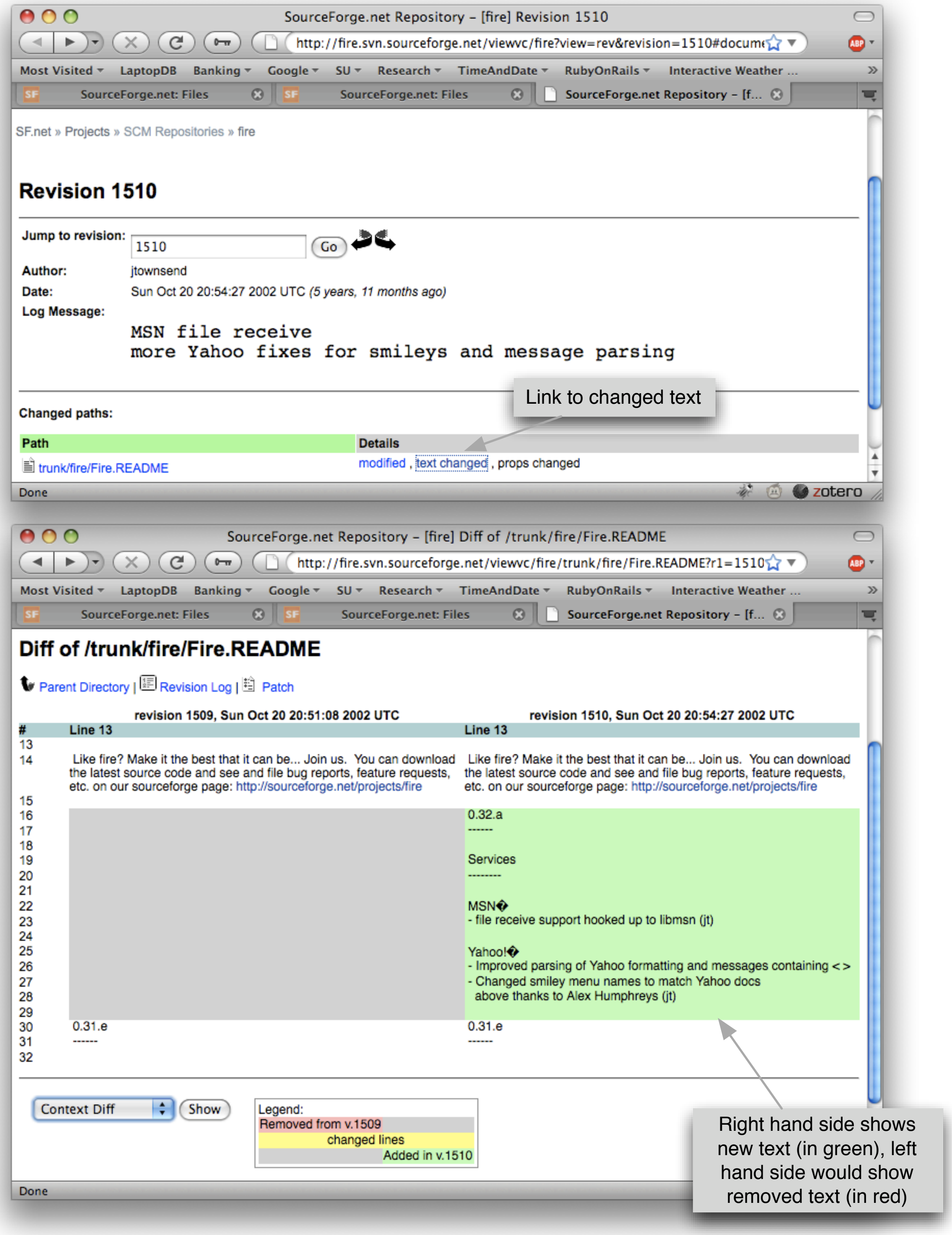
Figure 5.6: The Release Notes show one view of the Tasks of the period. Note the bullet point division, the grouping headings, the developers initials following the item and, in a few cases, links to Tracker items.



Release Name: Fire.app 0.32.a

```
Notes:
0.32.a
------

We now include updated French, German, Icelandic, and Italian localizations.

Application
-----------

- Chat windows scroll the upper scroll view when PgUp/PgDown are pressed. (nk)
- Chat windows only auto-scroll if the last line of text is visible. (nk)
- Chat window preference to indent wrapped lines. (nk)
- Reorganized the menu layout. New Buddies menu makes it easier to add new buddies
  or access chat history, edit, etc. Internet related items moved to the Help menu. (jt)
- Away message counts reset when your status changes (feature 527207) (jt)
- Change the updates of the Buddy List.  This gets rid of most of the pauses in typing (6 hours fixing th
- Properly log messages from users with slash '/' characters in their name (bug 478127)
  Thanks to Dan Scully for the fix (ds/jt)
- Logging is now colorful and everything (gb)
- Clarified dialog box when sending to an offline buddy (bug 620848) (ah)
- Disconnect from all command in Dock menu (feature 587318) (jt)
- Refresh Buddies command in Buddies menu (feature 540353) (ah)
- Renamed Groups are are also renamed on the server (ah)
- Removed Groups are also removed on the server (ah)
- Drag and drop to move buddies & groups in the Buddy window now works correctly (bug 645016) (ah)
- May have completely fixed the no send on Enter/Return bug.  Thanks to Justin Wood for advice as to how
- Buddy list is in upper left corner by default. Send on enter is on by default. Flashing chat windows ar

Services
--------
AIM�
- This is the BIG one:  Oscar support! Don't go too nuts over this! (gb/special thanks to Rob Flynn and S
   Direct IM support when started on other side (starting it comming soon)(feature 613011)(gb)
   Away message reading support (feature 478146) (gb)
   .mac buddy support (feature 563733) (gb)
   Server Side Buddy list support with groups (more options on this comming soon)(gb)
   Buddy info is more accurate on login times (bug 614077) (gb)
   Invisible (even though it is pretty stupid) (gb)
```

Figure 5.7: The README file is an evolving record of the Tasks, written by participants as they provisionally complete work. This figure also shows the archive of a CVS check in, including the log message, the list of changed files and the ability to link to a detailed 'diff' showing actual changes.


SourceForge.net Repository – [fire] Revision 1510
http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1510#docume
Most Visited LaptopDB Banking Google SU Research TimeAndDate RubyOnRails Interactive Weather ...
SourceForge.net: Files
SourceForge.net: Files
SourceForge.net Repository – [f...
SF.net » Projects » SCM Repositories » fire
Done
zotero
Link to changed text

Revision 1510

Jump to revision:	1510 Go
Author:	jtownsend
Date:	Sun Oct 20 20:54:27 2002 UTC *(5 years, 11 months ago)*
Log Message:	

```
MSN file receive
more Yahoo fixes for smileys and message parsing
```

Changed paths:

Path	Details
trunk/fire/Fire.README	modified , text changed , props changed

SourceForge.net Repository – [fire] Diff of /trunk/fire/Fire.README
http://fire.svn.sourceforge.net/viewvc/fire/trunk/fire/Fire.README?r1=1510
Most Visited LaptopDB Banking Google SU Research TimeAndDate RubyOnRails Interactive Weather ...
SourceForge.net: Files
SourceForge.net: Files
SourceForge.net Repository – [f...
Parent Directory | Revision Log | Patch
Context Diff
Show
Done
Right hand side shows new text (in green), left hand side would show removed text (in red)

Diff of /trunk/fire/Fire.README

#	revision 1509, Sun Oct 20 20:51:08 2002 UTC — Line 13	revision 1510, Sun Oct 20 20:54:27 2002 UTC — Line 13
13		
14	Like fire? Make it the best that it can be... Join us. You can download the latest source code and see and file bug reports, feature requests, etc. on our sourceforge page: http://sourceforge.net/projects/fire	Like fire? Make it the best that it can be... Join us. You can download the latest source code and see and file bug reports, feature requests, etc. on our sourceforge page: http://sourceforge.net/projects/fire
15		
16		0.32.a
17		------
18		
19		Services
20		--------
21		
22		MSN�
23		- file receive support hooked up to libmsn (jt)
24		
25		Yahoo!�
26		- Improved parsing of Yahoo formatting and messages containing < >
27		- Changed smiley menu names to match Yahoo docs
28		above thanks to Alex Humphreys (jt)
29		
30	0.31.e	0.31.e
31	------	------
32		

Legend:
- 🟥 Removed from v.1509
- 🟨 changed lines
- 🟩 Added in v.1510

they consider the task provisionally complete. So the README file, or rather changes to the README file, becomes a second emic source of Task outcomes, providing a further 31 task outcomes ($\approx$ 29%).

Finally, reading other archives in the context of the Release Notes and README, makes it clear that not all work is considered worth adding to a README file, let alone announcing in the Release Notes. In these cases an analyst has to rely on a close reading of the CVS log messages, which are often very, very context dependent. The participant observation experience in BibDesk showed that such messages are literally "bashed out" while checking into CVS; they are additional, often annoying, work for the developer and their readership status is unknown. Therefore an analyst attempting to make sense of the archives must sometimes dig into the changes in the source code itself, using the link as shown in Figure 5.7. This type of recognition provided 10 Task outcomes ($\approx$ 9%).

Therefore the identification of Tasks began with the identification of a Task Outcomes by observing how participants themselves communicated the names and descriptions which they use to make sense of their work. Participants, of course, also understand the coherence of their work at various levels, as shown by the headings in the README and Release Notes documents, shown in Figure 5.6. These headings often evolve through the README file, or are added at release time, as a way of organizing their sub-points. While these structures are interesting, this work pushed towards the lower level descriptions of work, most simply by choosing the most indented lines in the release notes.

This section of work generated two new sense-making concepts to be added to the four outlined above: Task and Task Outcome. Task Outcome is the actual change to the group's shared outputs which occurs as a result of the work directed towards it. Thus a Task is created as a container, anchored by its Task Outcome, into which evidence about contribution towards the outcome could be placed.

I worked first through the Release Notes to create the first set of Task Outcomes, generating RDF statements describing each and including interpretative memos, as well as Task containers. At this stage it was obvious that some lines in the release notes were fairly specific, while others had brought together many smaller tasks which would later need to be broken out, based on finer grained evidence in the README and SVN log.

Organizing Documents according to Task

Once the analysis had recognized an initial set of tasks, I began reading the Documents and examining them for evidence as to which Task the activity they represent may have contributed. This process worked through the temporally ordered Documents and generated additional RDF statements that a particular Document was relevant to a particular Task. This was accomplished by free text searching through a full text output of the Documents, rather than searching directly in the database. This approach was simpler than writing and running queries and enabled ne to examine Documents nearby in time quickly. The process worked primarily with keywords from the Task Outcome, but also with synonyms and related concepts based on my growing knowledge of the codebase as well as general programming knowledge.

I was aided in this process of discovering relevant Documents by a short script that printed out the Tasks, their Task Outcomes and the Documents currently considered relevant, allowing me to assess the temporal and narrative coherence of the Task. Often this exposed logical gaps, such as a thank you message without evidence of work, which gave me new 'leads' to search the Document collection. In the RDF dataset (recall that this is a plain text file) I added an Object type which was a DocumentRelevanceAssignment, which linked a Document with a Task, stated at what time the assessment of relevance had been made and allowed me to record a relevance memo which captured why I thought this particular Document to be relevant.

For example, Fire Task 5 was generated from a line in the release note which read "Chat windows scroll the upper scroll view when PgUp/PgDown are pressed. (nk)". As I searched for relevant documents I recorded an Episode Memo describing my searching strategy, "Searched Scroll, and everything associated with nk, who is Nick Kocharhook, or nkocharh". This resulted in locating a set of Documents, including an SVN message with the log message, "Chat windows scroll the upper scroll view when PgUp/PgDown are pressed." As I designated this as related I recorded a Relevance Memo, "Clear svn connection; also this is an edit of the README file, which is clearly the basis for the change notes (although not everything makes it into the change notes)" (One can see my growing understanding of the role of the README file here as well). I also included the immediately following SVN document, based on a reading of the code changes. I continued to search for related discussion, such as a bug report or email thread, but in this case did not find anything that discussed scrolling or PgUp/Down.

All the time I was reading the Documents throughout the set, both assigned and unassigned, and comparing them to the full set of Tasks seeking congruence and coherence. This process is the 'shaking of the mobile' described above allowing the underlying structure to reveal itself, guided by theory and experience. The work was iterative, sometimes frustratingly so, as new evidence generated interpretative tensions which required either splitting or merging Tasks, and then re-working portions of the Document assignments to accord with these new anchors. Such re-working was memoed in the fields provided.

Often these discoveries involved time-scales of analysis and 'unpicking' the summarization in the Release Notes. For example, in Figure 5.6 under the title of AIM there is a note saying, "This is the BIG one! Oscar Support!" followed by a set of indented notes. Initially I understood this as a single Task, the addition of a library providing Oscar (a particular network protocol) support, together with a more detailed explanation of what that meant. As I became more familiar with the data I saw this differently, with the initial add of the library one task and each individual bullet point becoming a separate Task. This decision was made because it better made sense of the work; for example there was a substantial gap in time between the addition of the library and work on individual features made possible by the library. In particular some features of the library, and what they made possible, were only grasped and worked on once the library had been in place for some time. The library made them possible but was not put in place in order to achieve them.

Not all Documents were relevant to Tasks, as defined as work resulting in changes to the shared outputs of the project. For example on the user list there are many Discussions where users are seeking support in using the software, rather than reporting bugs or requesting features. Often the answers to this are relatively simple and provided by other users. Sometimes, however, the discussions do lead to the identification of a bug or the request of a feature which are linked to an eventual change (a Task Outcome). These discussions were included as relevant to a Task.

The only Document type where it is reasonable to expect that eventually all will be relevant to a Task is the SVN log messages, since they are directed linked to changes in the shared output. I therefore identified these as an indication when sufficient work had been completed. From time to time I ran a script which indicated how many SVN messages remained outstanding, and concentrated on these. In a small number of cases (only 10) these produced new Task Outcomes, which had not been considered substantial enough to include in the Release Notes. Eventually all the SVN Documents were assigned to Tasks, sometimes just made up of that one Document. I then worked through the dataset, using the script which showed Tasks and the Documents assigned to them in temporal order.

This phase of the analysis took close to four weeks of work, approximately two weeks for Fire and two weeks for Gaim. By its conclusion I was confident that I had made more sense of the Document collection than a casual reading of individual archival records; I was capturing my emerging understandings in an appropriate format and memoing frequently. The latent structure of the data was asserting itself and while alternative interpretative paths would have been possible, that structure latent structure would draw interpretation towards it.

Recognizing contribution

At this stage the dataset consisted of Tasks, each with Outcomes and relevant Documents. Yet the research question calls for a deeper interpretative understanding because it is about patterns of contribution. Documents provide evidence about activities that contribute, but they are at best a delivery mechanism for the contribution, and provide evidence about real-world activity. It was also becoming clear that some Documents provided evidence about activities which were not directly associated with the Document and which were not evidenced by any other Documents.

For example it was not uncommon to see a thank you note in a SVN check-in, such as Fire Task 3, where an updated Icelandic translation is checked in by jtownsend, along with the SVN log message, “Icelandic localization update from Heimir Freyr”. Here the Identifier on the Document (jtownsend) refers to the person that checked the translation in, but the message clearly conveys the knowledge that that person did not write the original translation. In fact it is clear that someone else (Heimir Freyr) did and provided the translation to the developer doing the check-in in some manner. I searched the archives to resolve this narrative tension but was often unable to find evidence of that transmission, so one can only assume that it was transmitted in a way not captured in the archives, perhaps by private email to the developer.

From the perspective of understanding patterns of contribution such tensions are very important, since it is clear that, although there is only one Document, there are two relevant activities by two different participants, albeit that they are playing slightly different roles.

Addressing these issues and resolving this narrative tension requires a new concept, which I called the Action. This is defined as the real-world activities contributing to the Task Outcome. These are distinct from Events (which generate Documents) and from Documents, which merely provide evidence about Actions. Actions are undertaken by Participants, and these Participants can be identified by multiple Identifiers. Recognizing the real-world people behind multiple Identifiers is also crucial to the research question, since individual work could easily be understood as collaboration simply because it is not realized that one individual is using two different email addresses.

Having generated this concept I worked through the Tasks, reading each Document in temporal order and created Actions as appropriate, memoing my reasoning for doing so. Simultaneously I assessed the Identifiers associated with the Documents or the content and created Participants to record who performed the Actions. I created two sub-classes of Actions, ExplicitActions and ImplicitActions. ExplicitActions were those where the Event and the Document provided direct evidence about the Action, such as a regular SVN checkin. For these one is able to take the time the Event occurred as the time that the Action occurred. ImplicitActions, however, are interpreted from the content of the Documents and therefore one must assess a time at which they likely occurred. For the most part one only knows that they occurred prior to the Event, and so I simply assigned them a time milliseconds before the Event. Sometimes, however, I was able to better locate them either through direct evidence ("last week") or through examining them in context and relying on narrative coherence.

Recognizing Participants was relatively easy. For the most part one can rely on data linking Sourceforge usernames to RealNames which are available in the Notre Dame dataset (in the Users table). This, combined with free text signatures in emails or Trackers (less common) lets one link between SVN commits and emails, for example. In a few cases one has to rely on interpretative clues such as the first part of an email address being the same as a Sourceforge username. For some participants, particularly those in ImplicitActions, one has to create a Participant not linked to a Sourceforge user account. The human experience with different forms of names performs very well in these situations, where automated techniques (which were also tried) had substantial difficulties.

Recognizing Actions from relevant Documents took approximately two weeks for both projects. This iteration sometimes revealed issues with earlier assignments or Tasks, which were corrected when observed. By this point I was quite familiar with the dataset and the emerging interpretations.

The set of concepts generated to this point can be illustrated through three examples.

1. A request for a feature sent by email (where the feature was eventually implemented)
2. A release of the software, and
3. A CVS log message that thanks someone other than the person doing the check-in for their work.

An emailed request for a feature (which is eventually implemented) is the simplest of our three examples. Here the content of the email is a Document, the author (and requester) is a Participant, their email address and real name are Identifiers, and the sending of the email is an Event. The requested feature is a Task Outcome (because it was eventually implemented). The request itself is one of the Actions that contributed to the Task Outcome. The full set of Actions that contribute is the Task. Note that here there is a one-to-one-to-one relationship between the Event (sending the email), the Document (the content of the Email) and the Action (making the request).

A release of the software is a little more complicated. A developer (the Participant) creates an updated binary of the application and fills out a form at Sourceforge. This form has three parts: the application binary for upload, a set of Release Notes and a Change Log. For some projects the Release Note and the Change Log are identical, but others use the Change Log to list changes, and the Release Note as a more discursive announcement. When the developer presses upload (the Event) this creates three separate Documents. These Documents could contain evidence about many different Actions contributing to many different Tasks (the changes to the application themselves). In the archival record the Participant in this case is identified by their Sourceforge User ID, the Identifier.

Just as the relationship between Event and Document can be one-to-many, so can the relationship between a Document and an Action. This is the case in the third example. A developer (the Participant) using his Sourceforge User ID (the Identifier) checks in a new translation and makes a log message (conceptually these are two separate Documents). The new translation itself is the Task Outcome (a change to the shared output of the project). However the Log message indicates that the developer doing the checkin did not write the translation himself, because he thanks another for it. Therefore these Documents provide evidence for two separate Actions, one by the developer doing the checkin and another by the Participant who wrote the Translation. Both Actions are contributions to the same Task Outcome, therefore they are part of the same Task.

For completeness it is worth noting that these concepts relate to the three organizing structures reported in Chapter 3: Sessions, Discussions and Tasks. Tasks has the same meaning as before. Sessions indicate the material fact that participation is bursty: it occurs in periods when the participant is awake, at a computer and paying attention to the project. All the Events resulting in archived Documents, therefore, take place during one of a Participant's many Sessions, even though some Actions might not (most do, but conceptually one could draw at a whiteboard or think in the shower). Of course not everything done in a Participant's Session is directed to the same Task; developers often work on more than one thing.

Recall that Discussions are patterns of call and response in venues where this is recorded, such as Email Threads and Tracker Items. A Discussion is therefore a set of Documents. Discussions can sometimes contribute to Task Outcomes and therefore their content often includes evidence for Actions. Further, a Discussion often contributes to more than one Task Outcome, so a Discussion is not contained within a single Task, but can "lie across" many.

Together these factors mean that even within a particular Task the Actions contributing to the Task Outcome cluster in time, with dense periods and periods without any relevant Actions. Sometimes the periods of time without any Actions can be quite long (weeks, months, even years in some cases involving Trackers) but these gaps do not mean that there are two Tasks, since all the Actions still contribute to the same Task Outcome. The dense periods of Tasks might be called Bursts, although this dissertation does not make use of this concept.

The careful reader will notice that the word 'episode' does not appear in this discussion despite its prominence in the BibDesk case study and the literature review in Chapter 4. Episode was not used because it could refer to several of the concepts outlined above without being out of synch with the literature, including the definition of Annabi et al. (2008). For example it could refer to Tasks, Discussions or separate Bursts within Tasks, all of which are processes over time linking behaviors. One must make the decision as to what type of episode one is interested in, for example Annabi (2005) dealt with Episodes of learning and Heckman et al. (2006) dealt with Episodes of decision making. In this case I was interested in episodes of production activity, which I called Tasks, to emphasize that this is one, concrete, type of Episode and to avoid confusion with Discussions or Bursts.

Recognizing types of contribution

An Action is something done by a Participant that contributed to changes in the shared outputs of the project, where the shared outputs are usually changes to the application or its supporting documentation. Therefore categorizing an Action is a matter of asking "in what way did this contribute to the output?"

A classification scheme was developed based on my understanding of how work gets done in FLOSS projects. The development of this scheme began during the study of BibDesk presented on page 59, where I allowed myself open coding to generate vocabulary to answer the question about the type of contribution (e.g. Glaser and Strauss, 1967).

I then returned to the literature to examine a number of coding schemes which were potentially relevant. For example I examined the literature on Speech Acts and Conversation Analysis. I found these schemes to be too general for my specific research question; they focus on general characteristics of speech and rely heavily on divining the purpose of the speaker. I also examined schemes more closely related to work processes, including the Dialogue Acts scheme of Winograd and Flores (1987). This came closer, but was tuned mostly to situations of negotiation, and the data did not seem to fit this scheme well enough.

I returned to the data and focused on inductively building relatively simple models of software production. I was influenced by two meta-models of behavior: the waterfall model of software development and models of information seeking behavior.

The waterfall model is related to rational models of Intelligence, Design and Choice (Simon, 1960) and posits five phases of software development: Requirements, Design, Implementation, Verification and Maintenance. Information Seeking behavior argues

Figure 5.8: Inductive classification scheme for Actions

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-structuting the infrastructure and accounting for work/credit. It also includes declaring bugs fixed, or Patches applied (closing Trackers) *A Call for Release and Creating a branch in SVN (eg gtk1-stable)*
Assiging 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). *Implementing a feature (not neccessarily check in, since could be checked in on behalf of someone else)*
Polishing Production Work	Smaller changes that improve Core Production contributions (typos, integrations etc
Documentation Codes	
Documentation Work	Work that documents the code, application or activities. Includes pointers across Venues (eg in a Bug Tracker saying that a Patch has been submitted*)* *A ChangeLog/README only SVN commit.*
Planning Work	Work that documents one's own future activities (others is more likely to be management work)
Supporting Codes	Work that supports the work of others
Use Information Provision	providing information about the code base. Or use cases & RFEs and bug reports. These are often about conveying a private situation.
Code Information Provision	Includes providing suggestions about 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 dev seeking more information from a peripheral member.
Testing Work	Testing application functionality. This includes requesting more information from users in bug reports.

that behavior is driving by a need to close a gap where the seeker believes they do not have enough information to proceed (e.g. Krikelas, 1983). I was also influenced by a desire to focus on the act of programming, as opposed to surrounding and supporting acts.

The inductive scheme that emerged is presented in Figure 5.8. The five categories of contribution are: 1) Management work 2) Review work 3) Production work 4) Documentation work and 5) Supporting work, where each category has a number of sub-categories. The description of the category should be read in relation to its contribution to the work of the specific Task to which it contributes, not the overall project. Thus Planning Work is restricted to specific plans for the Task which the Action is part of.

Two aspects of this classification warrant further explanation. Unlike Conversation Analysis schemes, this classification does not distinguish between a request and a response; this is because requests by themselves do not contribute to the work, only the effect of a request, such as the information provided by a response, do that. However for a holistic, sequential, understanding of the Task it is necessary to also code requests, so they are coded according to how their response would have contributed; for example a request for further information on what a user was doing when they hit a bug would be coded as "Use Information Provision", since that is the type of contribution the answer would give. Interestingly requests were only found where their responses would be a form of Supporting work. This is consistent with earlier findings that participants do not assign work to each other (Crowston et al., 2005).

The distinction between Core and Polishing Production work emerged during classification of Actions and is a distinction of potentially high importance to answering the research question of this study. The distinction emerged due to a tension that emerged in the analysis process. In general it is hard to estimate the time and effort expended by a coder simply from the final results, since the code, build, debug cycle is private and a hard-won, innovative solution is often quite short in its final form. Conversely, however, it is usually relatively easy to see if a CVS check in is a quick, low importance change, such as fixing a typing error in a dialog box, or removing a compiler warning (not an error). As I sought to make narrative sense of the Tasks these small changes seemed qualitatively distinct from the larger coding contributions. They improved the 'shine' or release readiness of a core contribution, without fundamentally altering its functionality or behavior. As we will see below this type of work almost always *follows* a core contribution—as polishing follows construction—and when it is done by another coder is usually done without visible discussion.

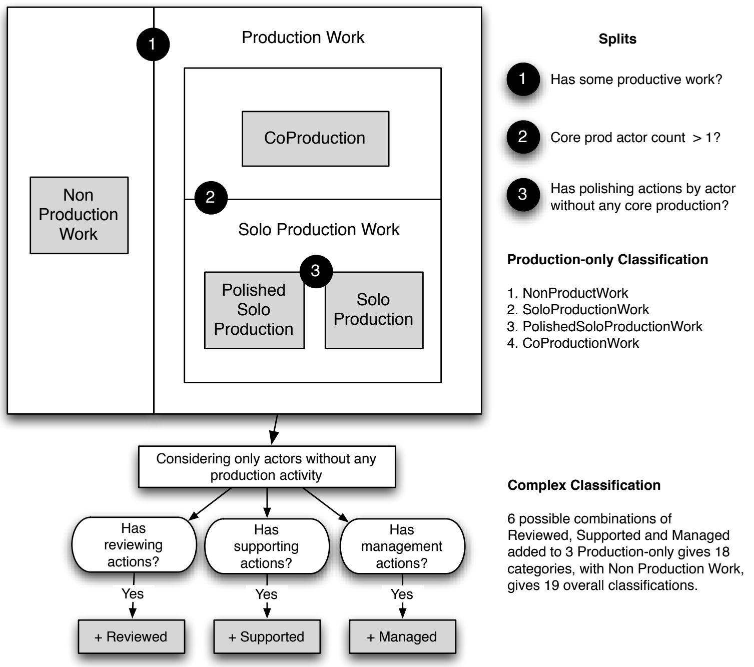
Nonetheless I was concerned that this distinction was driven by my working hypothesis of individual work. It was possible that I was discounting smaller contributions to defend the image developed in BibDesk of a single programmer leading the way in each Task. To avoid this I split the Classification of programming work into two categories: Polishing and Core and I took particular care to present and analyze the results both with this distinction and without, treating all programming work as a Core contribution. As we will see this distinction turns out not to be highly consequential.

To record the classification of Actions, I created another object in my dataset: a CodeApplication. A CodeApplication links an Action with its classification according to the contribution it was understood to make towards the Task Outcome. Also recorded are the time the classification was applied and a memo field to record my reasoning for applying the classification. The classification process was relatively speedy, taking about 2 days in total. The reason that this was quick is most likely due to simplicity of the classifications and the Tasks being organized into cohesive temporal narratives by the creation of Actions in the previous analysis step.

Classifying Tasks by contribution patterns

The detailed, iterative, qualitative work described above yields a dataset which is well-structured and amenable to automated classification according to patterns of contribution by different Participants. This classification gets at the main research

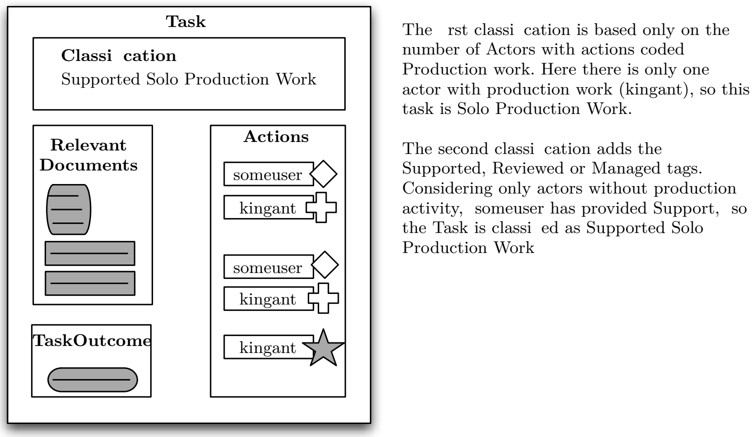
Figure 5.9: The workflow for classifying Tasks. The classification was performed programmatically


Production Work
1
CoProduction
Non Production Work
2
Solo Production Work
3
Polished Solo Production
Solo Production
Splits
1 Has some productive work?
2 Core prod actor count > 1?
3 Has polishing actions by actor without any core production?
Production-only Classification
1. NonProductWork
2. SoloProductionWork
3. PolishedSoloProductionWork
4. CoProductionWork
Considering only actors without any production activity
Has reviewing actions?
Has supporting actions?
Has management actions?
Yes
Yes
Yes
+ Reviewed
+ Supported
+ Managed
Complex Classification
6 possible combinations of Reviewed, Supported and Managed added to 3 Production-only gives 18 categories, with Non Production Work, gives 19 overall classifications.

question of this chapter, enabling a distinction between Individual work and other collaboration patterns. As we will see the classification works on two levels in order to provide a nuanced answer to this simple research question. The classification system developed is shown in Figure 5.9 (on page 115).

The classification proceeds by building up a name piece by piece. It begins by focusing on the two types of Production Work, choosing between Non, Solo, Polished or Co. Non Production Work are those Tasks where no Actions could be identified that contributed to the shared output of the product. These are all Tasks which were to do with administering CVS, either creating a branch or editing the .cvsignore file as a convenience for developers. These are potentially useful actions, but they don't affect the final shared work product, so they are dropped from the analysis at this point. There were three overall, all in Gaim. Each involved only a single actor.

Figure 5.10: The Task can be classified based on patterns in the types of Actions undertaken by Participants



The rst classi cation is based only on the number of Actors with actions coded Production work. Here there is only one actor with production work (kingant), so this task is Solo Production Work.

The second classi cation adds the Supported, Reviewed or Managed tags. Considering only actors without production activity, someuser has provided Support, so the Task is classi ed as Supported Solo Production Work

Solo Production Work means that only a single actor performed all of the production work in the Task. Co Production work means that more than one actor contributed core production actions. Polished Production Work is a hybrid, and means that while only a single actor did core production work for the Task, one or more other actors provided polishing tweaks to their work. As discussed above, the distinction between polishing and core work, while clear to me, contains an interpretative tension. It is possible to argue either way for whether these show interdependency in work and therefore they are separately classified. As we shall see, there are relatively few and wether they are grouped with the Solo or Co Tasks does not substantially affect the results.

The second step in classification adds nuance and only considers Actions performed by Participants who did not also perform Core Production Work. It then adds a word depicting the type of additional work these Participants did, whether that be Reviewing, Supporting or Management work. Dropping core production participants before this addition ensures that the classification is interpretable as one about collaboration; a self-managed or self-supported task is, for our purposes, simply Solo Production Work.

The classification can therefore be interpreted at two levels. The first is only in regard to Production Work: Non, Solo, Polished and Co. The second, more complex, classification may add up to three additional labels to the three types of Production work, describing whether the production work was Reviewed, Supported or Managed. On further inspection almost all the management work simply consisted of closing Tracker items, rather than the more interesting codes for classifying task assignment etc. This is consistent with earlier findings that FLOSS participants do not assign tasks to each other (Crowston et al., 2005). Therefore, for clarify of presentation, the additional Management tag was dropped in the presentation of results below.

5.3 Results

We can now present the results of the study, beginning with broad descriptive outlines of the set of Tasks, then presenting the result of the classification and presenting detailed illustrative examples of each type. Finally we examine a set of Tasks which provide evidence of productive deferral.

Descriptive Summaries

The analysis of the two periods yielded 106 tasks in total: 62 from Fire, 44 from Gaim. They can be summarized in a number of ways: the number of actions that contribute to them, the number of individuals that contribute and their duration. The shortest tasks, in terms of action count, were only a single action, while the longest, an outlier, included 81 actions, of which 34 were coded as Production Work

Figure 5.11: Distribution of Actions per Task. Note that an outlier of 81 Actions was removed for this figure

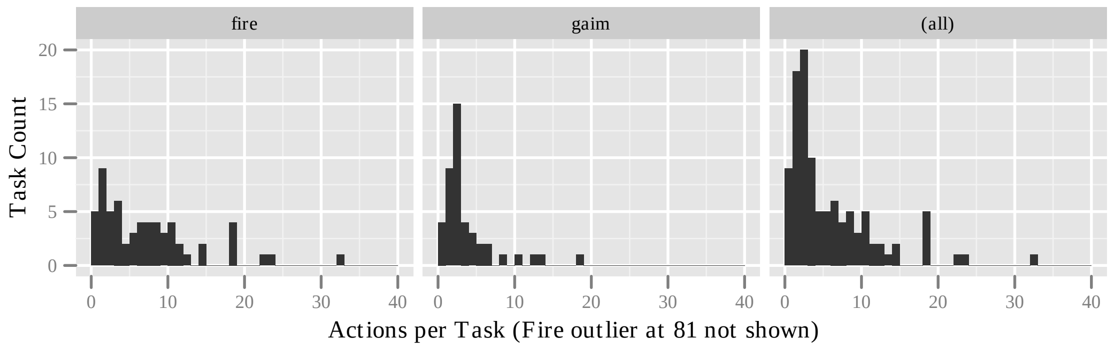
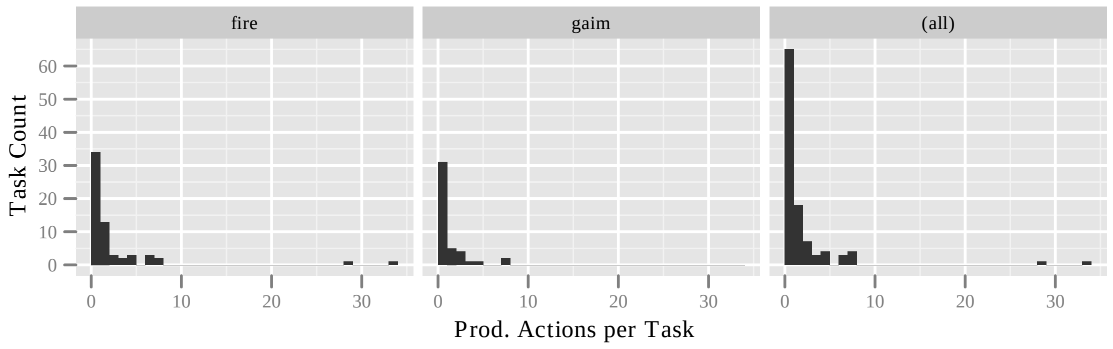


Figure 5.12: Distribution of Production-only Actions (core and polishing) per Task.



(by 2 separate actors), as shown in Figure 5.11 (page 118) and Figure 5.12 (page 118).

In terms of Actor involvement, there were 30 tasks with only a single actor, while the highest overall Actor count was 11 (again in the outlier). Considering only production actors the range was only from 1 to 3 actors, heavily skewed towards 1. These distributions are shown in Figure 5.13 (on page 119) and Figure 5.14 (on page 119), where the skew of the data towards lower actor counts is clear.

Figure 5.13: Distribution of distinct actors per task, considering all Action types.

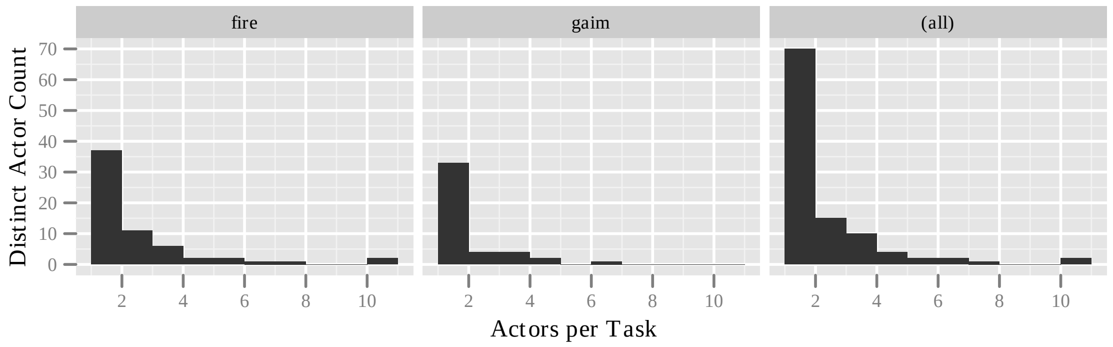
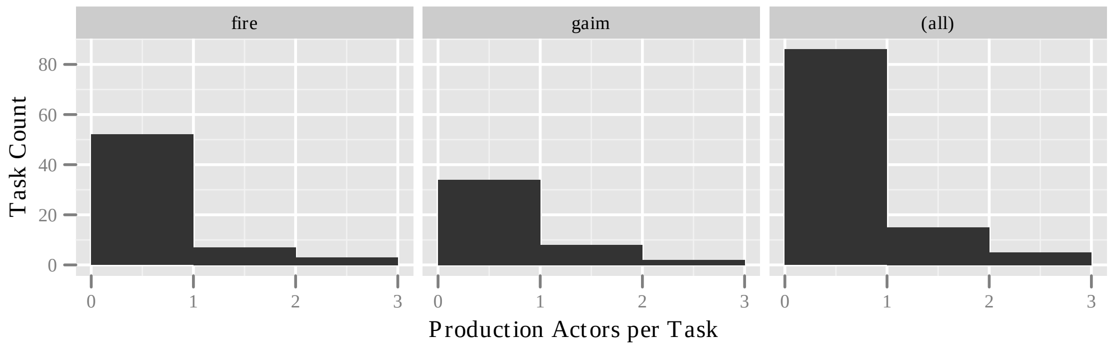


Figure 5.14: Distribution of distinct actors per task, considering only Production actions



In terms of calendar duration (from first to last Action), the shortest tasks were instantaneous, since they consisted of only a single Action. The longest, including all Action types, was over 413 days. Considering only Production actions, the longest Task was 67 days. Some Tasks were longer in duration than the inter-release periods since they included Actions from Trackers or Email threads that were outside the period; all Actions coded "Production Work" occurred inside the period. These distributions are shown in Figure 5.15 (on page 120) and Figure 5.16 (on page 120),

Figure 5.15: Distribution of temporal period from first to last Action (of any type)

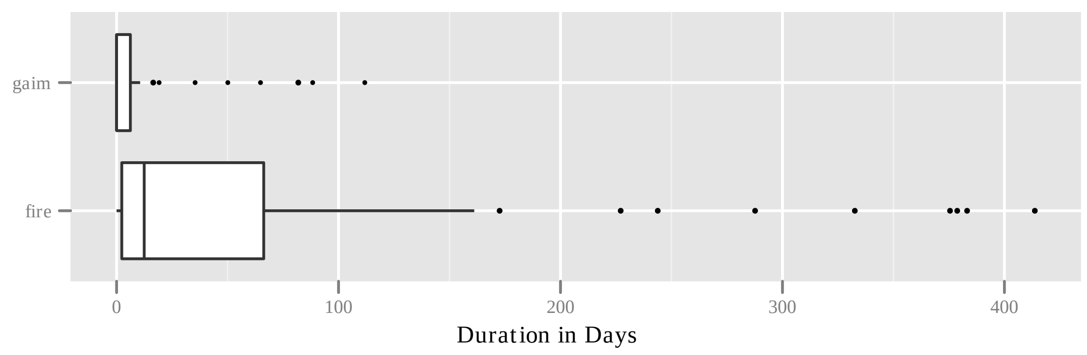
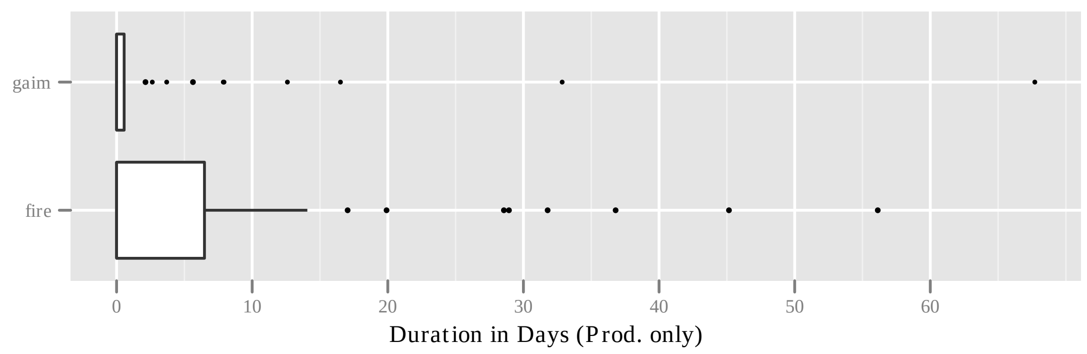


Figure 5.16: Distribution of temporal period from first to last Production-only Action (core or polishing)



using boxplots showing the mean and bulk of the distribution below 2 weeks, with long outliers (which are discussed in detail below).

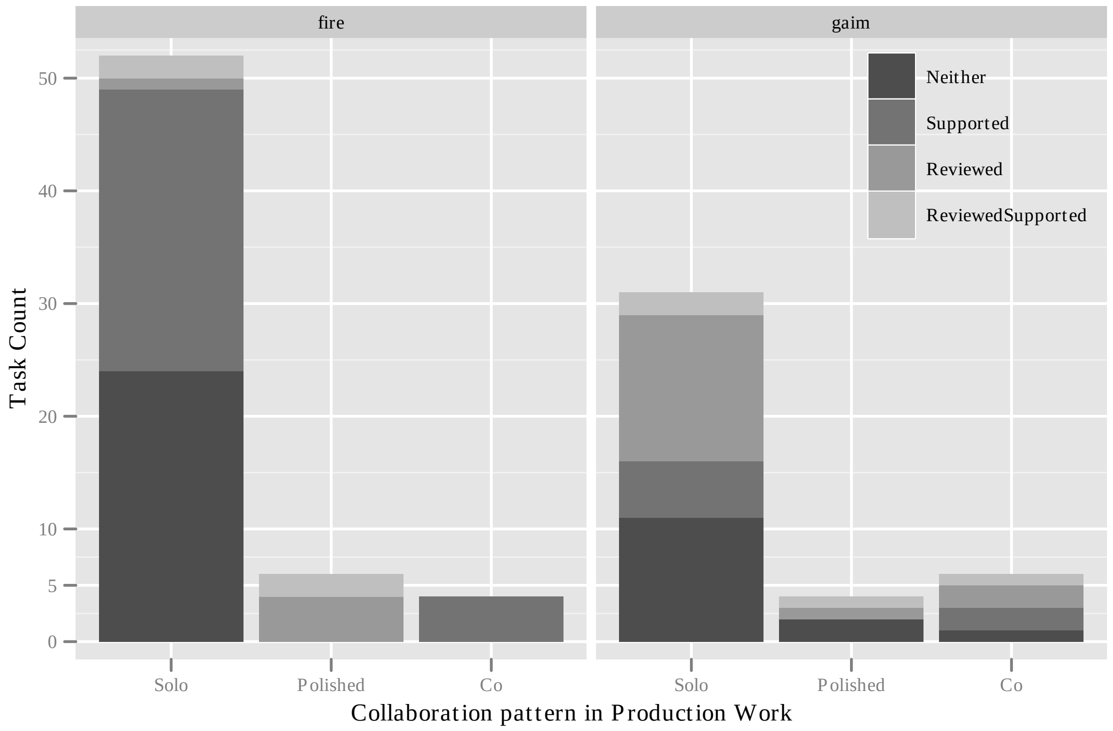
Classification Results

Figure 5.17 (on page 121) and Table 5.2 (on page 121) show the results of the classification. Overall, of the 106 there were 3 Non production work Tasks (all in Gaim),

Table 5.2: Collaboration Classification

	Non	Solo	Polished	Co	*Sum*
Fire	0	52	6	4	*62*
Gaim	3	31	4	6	*44*
Sum	*3*	*83*	*10*	*10*	***106***

Figure 5.17: A figure showing the results of the classification. The basic classification is shown across the X axis, task counts on the y axis. The stacking in the bars shows part of the complex classification, based on whether the Tasks also had contributing Reviewing or Supporting actions. The dominance of individual work is clear in the dominance of the darker sections.



83 Solo Tasks (Fire: 52, Gaim: 31), 10 Polished Tasks (Fire: 6, Gaim: 4) and 10 Co Production Tasks (Fire: 4, Gaim: 6). This shows the dominance of the Solo production mode: fully 81% of the Tasks were the result of individual production efforts, against 10% where two developers provided substantial contributions. Even if one drops the distinction between core and polishing production work, thereby moving all Polished tasks to Co work, the dominance of individual Tasks remains, with 83 Solo tasks and 20 Co work tasks.

The additional classification shows that fully individual solo work is the predominant collaboration mode, while providing additional subtlety. Table 5.3, and the dominance of the darker sections in Figure 5.17, shows that the largest single category was Solo work that was neither reviewed nor supported by other actors in any way (35), although this is true overall all only because it is strikingly true for Gaim.

Table 5.3: Solo-only Additional Classification

	Neither	Supported	Reviewed	Rev. & Sup.	*Sum*
Fire	24	25	1	2	*52*
Gaim	11	5	13	2	*31*
Sum	*35*	*30*	*14*	*4*	**83**

Supported Solo work was a close second with 30 total tasks, dominated by 25 such tasks in Fire. The majority of these reflect user reported bugs or requested features that were eventually implemented by a single developer. Only a handful of tasks involved active testing of debug or CVS builds, in many cases these supporting reports or interactions were quite old (100s of days), and in some they reflected additional user reports of bugs against the version “in the wild”, but already fixed in CVS (i.e. the Actions coded supporting follow the Actions coded Production.). Most of the Reviewed tasks involved the creation of a translation which was checked in by a developer with CVS access, although Gaim incorporated more non-translation user patches than did Fire.

Illustrative Tasks

This section presents an illustrative task for each of the categorizations presented above, in order to give the reader a richer feel for the data, and lay the groundwork for illustrating the theory developed in Chapter 7. It is clear that not all similarly classified tasks were identical to those presented here, so where possible characteristic variations for each Task type are noted. The presentation begins with the most collaborative, *and least common*, type of task and works its way to the least collaborative, *and most common*, types of tasks. The full output for these tasks, including links to source Documents, is presented in the Appendix, on page 230.

Illustrative Co-Production Work

Table 5.4: Illustrative Co Work

#	Date/Gap	Actor (overall role)	Action	Code Applied
		Gaim Task 2: manual browser security fix		
1	Jul 20 2002	kareemy (user)	reports bug	Use Info. Provision
2	1D 5h 50m	lschiere (dev)	attempts diagnosis	Code Info. Provision
3	(undated)	robot101 (p dev)	writes patch	Core Production
4	20D 9h 41m	seanegan (dev)	checks in patch	Review
5	10D 18h 10m	seanegan (dev)	tweaks fix	Polishing Prod.
6	1D 20h 8m	chipx86 (dev)	re-writes fix	Core Production
7	1D 3h 20m	seanegan (dev)	move fix to branch	Management Work

In late July/early August 2002, the Gaim project had a potentially serious bug: the 'manual browser' setting could allow an attacker to gain additional privileges and delete files on a Gaim user's computer. It was a serious but not uncommon type of bug; the result of not properly sanitizing a user's input. The Actions contributing to the Task are shown in Table 5.4.

A user reports the bug, and a developer responds relatively quickly with a diagnosis (which turns out to be wrong). Two weeks later a peripheral developer writes a patch and submits that to a core developer, who checks it in and documents the change in the README file and the SVN check-in log. Ten days later, without intervening discussion or testing, the core developer returns and tweaks the fix. A day later, again without discussion, a different core developer rearranges the fix entirely, using a much more robust technique. Finally the developer preparing the release, on a branch, moves the improved fix to that branch. In this way the fix makes it to the 0.59.2 release and is a bullet item in the release notes.

The task involves seven separate actions by five distinct participants, three of whom make substantial changes to the code base. This task is shown in Table 5.4 (the undated action is an ImpliedAction, based on seanegan explicitly thanking robot101 for a patch as he applies it, the exact time the work was done is unknown, but it is known to have happened prior to its review and check in).

This Task illustrates a popular image of how FLOSS collaboration works: input from the community and multiple developers helping each other out. It is striking for doing so in a short task, but more striking as one of only two examples of this style of collaboration in the dataset. The other is the Fire project working to integrate a new version of the libfaim library, providing access to the OSCAR protocol for the first time (it is also the longest, at 81 actions and 11 distinct participants). While this image of highly collaborative work is inviting and matches expectations regarding teamwork it is far from common.

Table 5.5: Illustrative Polished Work

#	Date/Gap	Actor (overall role)	Action	Code Applied
		Gaim Task 34: TOC and ICQ plugin removed		
1	Aug 07 2002	seanegan (dev)	removes old code	Core Production
2	18h 16m 35s	chipx86 (dev)	reverts error	Polishing Work
3	1D 9h 47s	chipx86 (dev)	fix remaining bug	Polishing Work

Illustrative Polished Work

The illustrative Task for Polished Production Work is work done to remove an out of date feature in Gaim, see Table 5.5. It is notable because, while the core work is done by one developer, another developer, without speaking with the first, fixes a small error in the first developer's code. The second two actions are coded as Polishing Production Work, because they are small and merely complementary to the earlier contribution.

This Task is far more indicative of the full dataset than the Co work task shown above. The developers act without discussing their work before hand, and do not announce it on the mailing lists or Trackers. The polishing here is a very small amount of work, but it is important (the first polish actually fixes the build).

Illustrative Solo Work

The bulk of the work tasks done in the periods studied are classified as Solo work, just under half of which is Supported in various ways, but where only a single Participant is actually changing the software code. Since these are dominant and small, I have chosen one of each type to illustrate this class of work, as shown in Table 5.6.

There are 30 tasks that only involve a single actor, and 9 of these only involve a single Action (the others range from 2 to 8 in length, with 4 longer than 3 Actions). An example is 'user list duplicate fix' (Fire Task 57), where gbooker, a core developer, makes a single check-in that fixes a situation in which users were showing up twice

Table 5.6: Illustrative Solo Work

#	Date/Gap	Actor (overall role)	Action	Code Applied
		Fire Task 57: user list duplicate fix		
1	Dec 06 2002	gbooker (dev)	fixes bug	Core Production
		Gaim Task 3: iconv library integrated		
1	Aug 02 2002	seanegan (dev)	adds library	Core Production
2	19m 52s	seanegan (dev)	changes ChangeLog	Documenting Work
3	26m 10s	seanegan (dev)	integrates library	Core Production
		Fire Task 5: scroll on PgUp		
1	Nov 19 2002	nkocharh (p. dev)	makes PgUp scroll	Core Production
		Fire Task 29: AIM buddy icons		
1	Oct 27 2002	gbooker (dev)	checks in buddy icon code	Core Production
2	(same time)	gbooker (dev)	changes ChangeLog	Documenting Work
3	39m 3s	gbooker (dev)	add jpg icons	Polishing
4	1h 22m	gbooker (dev)	add bitmap icons	Polishing
5	12h 1m	gbooker (dev)	.buddyicon save	Polishing
6	1h 22m	gbooker (dev)	add bitmap icons	Polishing
7	3D 13h 1m	gbooker (dev)	fix IRC icons	Polishing
8	3D 18h 34m	gbooker (dev)	fix memory leak1	Polishing
9	1h 6m 23s	gbooker (dev)	fix memory leak2	Polishing
		Gaim Task 18: Finnish Translation (Reviewed Solo)		
1	(unknown)	teroajk (trans.)	writes Finnish trans.	Core Production
2	Jul 02 2002	robflynn (dev)	checks in trans.	Review Work
3	(same time)	robflynn (dev)	thanks teroajk	Management Work

in the user list. There is no indication in the archival record as to how he became aware of the issue; it may have been via private email, but it is also likely that he came across this small issue himself and, without seeking permission, planning or announcing his intentions, fixed the bug.

These Tasks are not necessarily trivial. A second example is 'iconv library integrated' (Gaim Task 3) in which seanegan integrates a new library which provides much needed character set conversion services, facilitating an improvement in handling non-latin characters. As shown in Table 5.6, it consists of three Actions, all by the same person (the third action brings together three separate CVS checkins occurring close in time and code; the two following are bug-fixes for the work of the first check-in). Table 5.6 also includes two additional straight solo tasks and an illustrative Reviewed Solo task.

5.4 Discussion

The results show two clear findings regarding patterns of collaboration. The first is that the organization of work, at a task level, is heavily skewed towards individual work, a theme across all the distributions above but most clear in Figure 5.17 (on page 121). Even when others are involved they are almost never doing concurrent development, but helping in others ways such as providing information about program use. Only 19% of tasks involved more than a single actor doing production activity, and in half of these the second actor only provided polishing-type production changes, such as typos or small leak fixes.

The second was that tasks tended to be relatively short in time, with the majority (65%) completely playing out over less than 2 weeks, shown in Figure 5.15 above and below in Figure 5.18. Given that some of the longest tasks include Tacker comments from previous years, the time compression is even more clear when considering just the span of production activity on the Task: only 10% of tasks extended longer than 2 weeks.

A reasonable objection to the results above is that this skew is seen because there are many simple tasks and relatively few complex tasks. Individuals can handle simple tasks, but the project cannot truly excel without facing up to complex tasks and these require inter-personal dependency and collaboration. This study did not attempt to measure the complexity of the tasks undertaken; simple measures such as changed lines of source code are always suspect and would give clearly wrong answers here. For example the largest sized changes are the introduction of new libraries, which is just the result of copying already prepared code from another location, leading to very large numbers of added or changed lines. The harder work, of course, is integration, which would change a much smaller number of lines, but reflect much harder work. For example, as a library is checked in, changing over 2,500 lines, one developed noted in the CVS comment, "Silly cvs... Most of these files were not changed..."[2]. In contrast a check in involving changes to only 22 lines included the comment, "I spent 6 hours traking [sic] this thing down!!!"[3] and another, adding 20 lines and deleting 30, "It was 15 fun hours of debugging!"[4].

The archives do not provide good evidence about complexity and task duration should not be used as a proxy for effort, since it tells us little about how much of the time elapsed between events was actually spent working on the Task. Certainly some of the Solo work is simple, but so too are some of the Co Work Tasks. Many of the Solo tasks appear far from simple.

Evidence for deferral

The second research question of this chapter asked whether there was evidence of deferral of work. There is good illustrative evidence for this aspect of FLOSS organizing. There were 17 tasks (16%) which had supporting actions which were outstanding for a period longer than the whole inter-release period (15 were longer than 100 days and four longer than a year). Clearly these were desired features, and in most cases the developers indicated that they desired their inclusion in the software (in one case the most active at the time was the initial requester). These tasks are shown graphically in Figure 5.18, on page 129, where different types of actions are shown with different shapes, and different actors in different shade of gray. Note the early Tracker supporting Actions (more pronounced for Fire than Gaim) and the intense activity

[2]http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1674#document

[3]http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1589#document

[4]http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1779#document

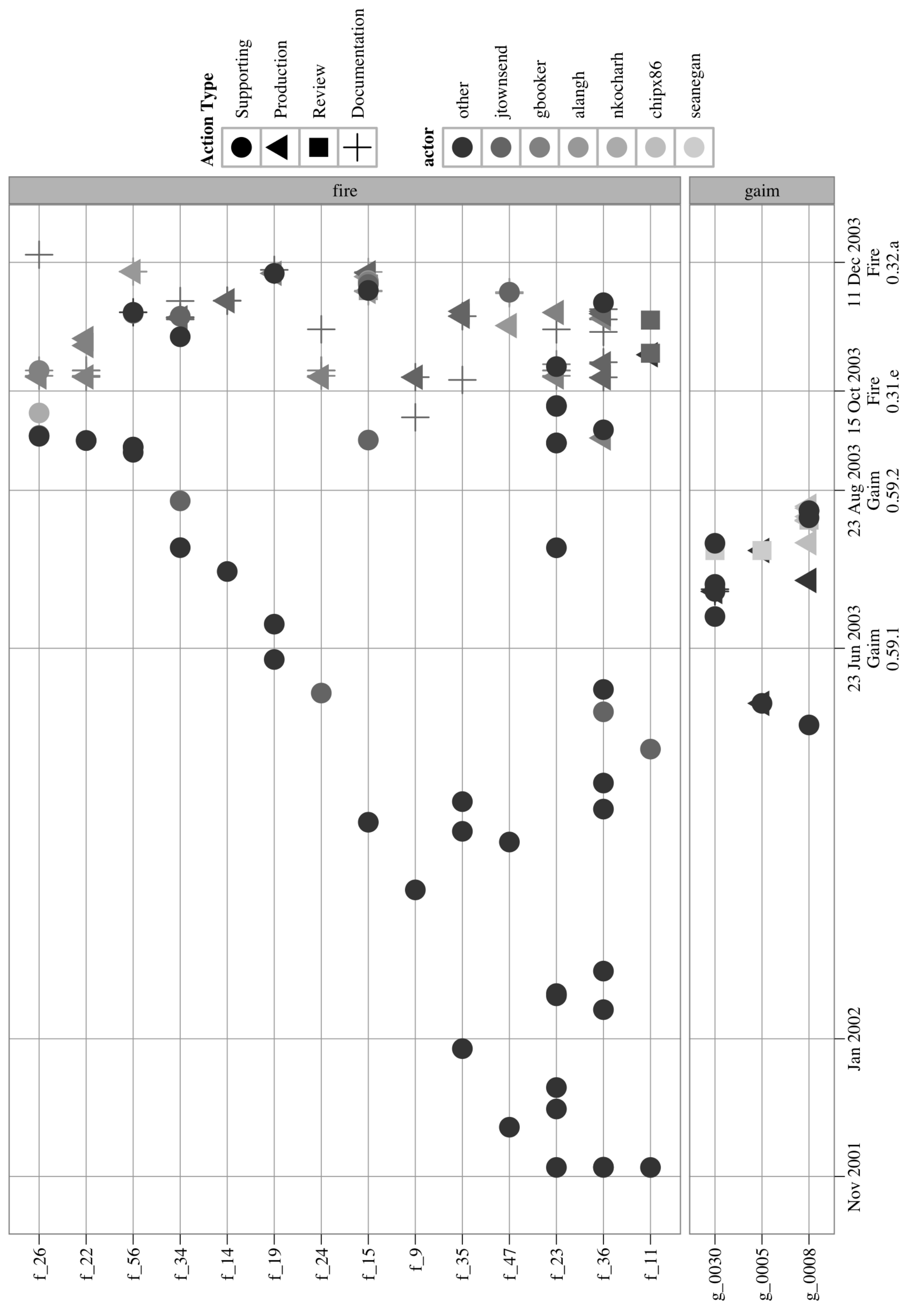


Figure 5.18: Tasks over 80 days in total duration. Each line is a task, each item is an Action with the coded type shown by shape and the actor by the shade of gray.

during the respective release periods (shown on the bottom axis). The darker shade of the majority of early Supporting actions indicates that they were performed by 'other', indicating active users not in the top seven actors by count. jtownsend, a core developer, also participates early, usually with diagnoses.

This pattern is not merely an artifact of the method, caused by not looking for early production actions, because the Fire developers return to the Tracker to comment when they are working on the task and often include comments like "by popular demand" or "that was an old one" in their documentation of their work. These tasks include protocol dependent things like adding file transfer, or away messages to Fire.

For example, in Task Fire 9 in March 2003, a user requests that the away message only be sent when it changes. Also in March 2003, one of the developers assigns the request to himself, indicating acceptance of this as a desirable feature. Yet nothing actually happens in terms of code production until October 2003 when jtownsend re-assigns the feature to himself and says,

> *This is possible now with the 'once' option for how often to send away messages. We just need to reset the message count when changing state....*
> *I think I have a fix for this... probably will check it in the next week or so.*

And the fix is checked in a few weeks later. This indicates well the way in which the project deferred the task until it was easier due to the unrelated implementation of a different feature.

Another similar pattern is relatively common in Fire, where work is deferred pending the arrival of new underlying protocol libraries. Task Fire 36 results in a working implementation of MSN file transfer. This feature was often requested, as early as November 2001, with regular requests and duplicate tracker items. In April 2002 jtownsend comments, "To get file transfer in MSN we need to upgrade to the 2.x version of the MSN library we're using." Various alternative options are suggested by users (e.g. posting ftp links) but the feature remains unimplemented until libmsn is upgraded and integrated in October 2003. jtownsend remarks,

> *I've checked in preliminary MSN file receive support. File send is also supported by the version of the MSN library we have, but there's some work to do in Fire to expose it.*

That work is eventually completed and Fire has a two-way MSN file transfer function.

Task Fire 35 shows a similar pattern. In December 2001 a user requests ssl capability in Jabber. The initial request is followed, in April 2002, by a number of other users also requesting this. However it is not until October 2002 that jtownsend posts saying, "OK, it looks like Proteus recently added this so we should be able to leverage their code." (Proteus is another OS X multi-protocol IM client, a competitor in some ways). He checks in working code in November 2002, specifically thanking "the Proteus team" in the edit to the README file.

Deferred tasks are as interesting for what was not done as for what was. The projects did not attempt to build an implementation plan, working either individually, or given the complexity of reverse engineering the protocols, in groups to build the needed code. Instead work was deferred, the issue noted and from time to time revisited in the Tracker, either for a reiteration of the need for the feature, or for a developer to diagnose, in very broad strokes, what might be needed to allow the participants to begin implementation work. In almost all of these cases the projects eventually implemented the features by integrating a new library for an underlying protocol (e.g. libmsn, libyahoo2 or libfaim). Integrating these libraries is not too complex, but they allow the project to 'jump forward', turning what had once been too complex for the project to attempt into work that can be accomplished through layers of relatively small, quick, individual tasks that characterizes the normal work of these FLOSS projects.

It is clear from Figure 5.18, and the presented Tasks, that this is more common in Fire than Gaim, which shows only three tasks with long deferrals (and even those are shorter). This may relate to the ability of the Gaim project to build not only its IM client, but also to release a widely used software library: libfaim. libfaim (later libprpl) provides the underlying protocol library for the AIM instant messaging protocol. Indeed the longest episode in the Fire project is the integration of a new version of this library which they specifically thank Gaim (and active individual developers). Clearly the Gaim project did not always defer difficult work, but understanding how they were built would take a detailed task-level study of those specific libraries (there was no work on the library in the period studied). It is possible that they themselves rely on libraries and were built layer-by-layer.

Limitations and Threats to Validity

There are four main limitations of this study. The first are related to case selection and were discussed above. The second involves questions about the place of this study in the overall dissertation. The third involves questions of the reliability of this study and the fourth relates to threats to the findings of deferral.

The role of this study

A reasonable objection to the design of the study is whether it can truly serve its intended role of replication in the overall structure of the dissertation. At issue is whether the fact of the analyst's experience and pre-existing conclusions from BibDesk do not undermine the replication purpose of the study. Why would the analyst be restrained to see what actually exists, as opposed to what he already believed to be present?

The study as reported above addresses this very real concern primarily by relying heavily on the participant's own organization of work into Tasks, through the evolving

README and the enactment of the structure in the release notes. This emic anchor is a fundamental starting point for the analysis, providing over 90% of the Task Outcomes. Secondly the category of polishing production work was created for those Actions for which I felt interpretative strain, and was therefore concerned that the hypotheses were influencing the decisions untowardly. By presenting these decisions as a separate category the robustness of the work was tested because the result remained even if these were counted as Co work. Finally, while Solo work was found to be dominant, a significant amount of Co work was also found and presented, showing the method and analyst to be at least capable of seeing this Task structure.

Reliability

A second reasonable objection to the overall method is to ask, if the research question is relatively straightforward, why is a qualitative, intensive effort the most appropriate method? Why not a standard deductive content analysis effort with multiple trained coders which is far more capable of demonstrating the reliability of the results? The answer is not that a study attuned to reliability is epistemologically impossible but rather than it would have been damagingly premature and would have undermined the theory development aims of the overall dissertation. The argument for this is in four parts.

Firstly, building a coding scheme based only on the experience in BibDesk would have been premature because I was not yet clear on how experienced task structure would be revealed through purely documentary inquiry rather than experienced participation. Replicating the qualitative study in these two projects was necessary to assure myself that these structures were indeed recognizable just from documents.

Secondly, I was not yet confident of my ability to convert tacit interpretive understandings into a deductive, rule-based coding scheme capable of being taught to those without field experience. If the original analyst could not replicate the findings certainly he could not teach the method to others. More importantly, the effort to systematize experiential findings for documentary analysis surfaced and made explicit my own understanding. This is shown through the generation of concepts such as Event, Document and Action, which were not present prior to the systematic sense-making undertaken in this chapter.

Thirdly, the iterative process of interpretation simultaneously creates the categories (Tasks) and assigns evidence to them (Documents and Actions). This is more complicated than a standard content analytic exercise in which the categories are deduced from the literature and the unit of coding (word, sentence etc) is uncontroversial. Again it would have been premature to attempt agreement statistics this complex before the constructs were made more clear through iterative sense-making.

The fourth, and most important, reason is that the process undertaken in this chapter created the space for the theory to be presented in Chapter 7 to develop. In particular the iterative and immersive method allowed the analyst to understand further the role of deferral and the systematic role it plays in facilitating largely individual work. An effort to focus too early on reliability alone would have hindered the overall theoretical goals of the dissertation. The method provided illustrative and generative understandings which guided the theoretical insights. Even if a coding scheme could have been generated without this sense-making work and shown to be fully reliable it would still only cover two FLOSS projects. For this reason the work reported in this chapter is more valuable for its role in generating and illustrating the theory presented in Chapter 7, and was therefore appropriate.

Overall, then, this study prioritized confirming the validity of the constructs and allowing theory to develop over and above testing the reliability of pre-determined constructs. This does not come without a price, of course, and the section below outlines the specific ways in which questions of reliability threaten the findings of the chapter. As a result of this study future research is now in a much better position to take up these challenges.

Considered from the perspective of positivist content analysis there are five separate coding decisions made in this analysis.

1. Recognizing separate Task Outcomes,
2. Assigning Documents to Tasks,
3. Recognizing Actions evidenced by Documents,
4. Recognizing which Participants undertook the Actions, and
5. Recognizing the contribution of the Actions to the Task Outcome.

If the concepts developed are robust and can consistently organize reality then multiple independent coders should agree on the coding decisions at each of these steps. It is, however, usual to allow for some relatively small level disagreement, perhaps caused by coder fatigue or a small set of either interpretative differences (akin to measurement calculation) or even a small amount of ambiguity in the concepts. This is why multiple coders are usually held to be in sufficient agreement when they agree in over 80% of the cases, after adjusting for random disagreement (such as through a kappa statistic).

The main result of this chapter, the dominance of individual work, is based on the number of separate participants contributing code production towards different Task Outcomes. It could be threatened by issues in each type of coding, although some more than others. Working backwards, only a small subset of Actions were coded as Production Actions, so as long as agreement was achieved on that code, disagreement between, say, Management or Supporting contributions would not be a threat. The category of Polishing Production was created for the cases in which the contribution was borderline, such as fixes to compiler warnings. Recognizing which Participants undertook a production Action is obviously fundamental, but is a relatively easy process, since it only involves either exactly matching unchanging Sourceforge usernames or recognizing multiple forms of Participant's real name, which is a natural skill of human coders.

Recognizing when an Action is evidenced in a Document is somewhat more complicated, but the vast majority of Actions are explicit, meaning that the existence of a Document (e.g. SVN check-in) already makes it clear that an Action has taken place (of 758 total Actions, only 39 were Implicit). Implicit Actions have to be recognized from reading the textual content of the Documents, but in the majority of cases this is triggered by the inclusion of a name and a phrase indicating credit or thanks. Simple rules seem likely to be able to cover these cases.

Recognizing when the content of a Document relates to a specific Task is a more complex decision, as shown in the examples presented above. It relies on an understanding of the Task Outcome and the software development required to accomplish such a Task Outcome. This requires some contextual knowledge of the overall applications and their interaction with things like the individual IM protocols. Introspection on the process through reviewing coding memos reveals that the most common factor for recognition is the identification of keywords and their reoccurrence in other related Documents. However in many cases one must not match the exact keyword (or a stemmed version) but a synonym or nearby concepts that, based on one's understanding of software development, are likely to arise in the course of work or discussion regarding this Task Outcome. Also vital to this process is a global understanding of the other Task Outcomes, since these provide the anchors into which the Documents must be sorted. Coders would be aided by the software developed in the course of this work which re-ordered assigned Documents into their real-time order, since this allows a coder to play "what-if" and to check the logical and narrative implications of a specific coding decision. There is also some assistance available from Discussion structures, such as tracker items and email threads. Tracker Items especially tend to be relatively focused, while email threads can split into multiple simultaneous topics, but the Discussion still has linear, if parallel, coherence.

Of probably the most importance is the recognition of specific Task Outcomes. These anchor the entire process, the Documents are held to be relevant to them and the Actions are coded according to an assessment of a contribution to them. The iterative process of assigning Documents and checking the narrative coherence of the Tasks may lead to changes in the set of Task Outcomes, which then cascade back to earlier decisions about relevance. The process is vastly improved by the line-by-line nature of the Release Notes and by observing the evolution of those notes in the README throughout the chosen period, which together provide over 90% of Task Outcomes. This provides an anchor which relies not on the coders perceptions but the participant's near-contemporaneous perceptions of the Tasks Outcomes they were accomplishing. Mechanistically separating the release notes into Task Outcomes would provide enough initial agreed structure that later disagreements about Tasks not mentioned in the Release Notes or README might have less impact. Simply following the rule that a Task Outcome is any line at the maximum level of indentation would provide automatic agreement of about 60%. Further, the existence of the initial list may provide sufficient common ground for the coders that other decisions become easier, especially when it comes to assessment of the README file.

Overall, then, it is important to recognize that the final set of Tasks and Actions ought to be in overall coherence and that individual coding decisions are likely to be increasingly influenced by an analysts' growing understanding of the whole period, as well as the working and documentation practices of the project. This makes testing each step as a separate 'coding scheme' problematic and suggests the best course for reliability checking would be to have two coders each code the entire period. This, of course, is substantially more effort than the common practice of agreeing on a subset of the corpus ($\approx 10\%$) then dividing the remainder between the coders. Having said that, it may be possible to check the reliability of some steps post-hoc by presenting new coders with a subset of the Tasks, together with all the Documents thought relevant to that subset, as well as a random sample of Documents thought to be relevant to other tasks (or none) and to test to see if an adequately similar structure emerges from an holistic understanding of the subset.

The experience reported in this chapter suggests that the riskiest steps are the first two, recognizing the Task Outcomes and finding the relevant Documents, particularly as the number of Task Outcomes rises and strains the analyst's working memory. If two Tasks classified as Solo tasks where the productive work was undertaken by different Participants were to be merged by arguing that their Outcomes were, in fact, the same, then that would reduce the count of Solo tasks by two and increase the Co tasks by one, very quickly affecting the overall ratio. A key aspect of coder training, therefore, would be training to understand the genre of release notes as well as general familiarity with the actual software application itself. This also suggests that time ought to be invested upfront in ensuring that the two coders generate close agreement in the set of Tasks, working just with the release notes and README file. As suggested above, simple near-automatic rules can provide substantial baseline agreement.

Threats to finding of Deferral

The secondary finding of this chapter, that projects defer work and seemingly do so when it is considered complex, rests on shakier ground that the primary finding. Firstly the concepts are less well defined. Deferral is defined as an acceptance of the desirability of a task, but a delay in the work needed to accomplish it. This must be distinguished from group-level disagreement as to the desirability of an outcome, or even disagreement as to the intention of the outcome itself (sometimes feature requests are simply hard to understand). Furthermore the suggested reason for deferral, that of complexity, is difficult to judge since it relies on understanding the internal cognitive judgements of individuals. These are sometimes documented, as shown by the qualitative evidence for deferral (on page 130) but often are not documented at all.

However, it should eventually be possible to develop a deductive content analytic scheme to recognize and distinguish between disagreement, uncertainty and deferral in discussions. Developing such a scheme would be enhanced by pursuing interviews regarding the topic during the development of the scheme.

The graphical evidence, on page 129, relies on the observation that supporting Actions, usually requests and/or diagnoses extend back much further in time than the productive actions. However it is possible that this is simply an artifact of admitting Documents outside the release period from Discussions, especially from Trackers which tend to be much longer lasting. Effort to improve the usefulness of this more quantitative evidence for deferral should concentrate on expending the coverage of other Document types, especially CVS/SVN logs, to ensure that preparatory work was not being undertaken at regular intervals outside the focal period.

5.5 Conclusions

Table 5.7: Archive Study Findings

#	Finding	Justification
1	Work in Fire and Gaim was individual by a factor of 8:1	Classification
2	There was evidence of complex tasks deferred until libraries made them easier	Long tasks

The study reported in this chapter aimed to replicate a core aspect of the BibDesk case study: the dominance of short individual taskwork. The finding was confirmed. Indeed it is focused and strengthened with an even more apparent skew towards individual work and a quantification of the dominance of relatively short durations. In addition further illustrative evidence was found for the deferral of complex work.

While it is possible that developers in Fire and Gaim work even more individually than in BibDesk is it perhaps more likely that the strengthening of the BibDesk results are a result of the differences in method. In the BibDesk study development contribution was represented by a very weak proxy: whether or not the Participant was an overall developer. This study shows that while a Participant might contribute to development in some Tasks, they are quite likely to act in ways that only make a non-Production contribution to other Tasks. This confirms the understanding that developers in FLOSS projects are not limited in their roles. In fact it seems more likely that roles are additive: A developer making core code contributions does not seem to stop providing Support or Testing other's work and is more likely to do things like closing Trackers than to assist others with production coding.

Further the BibDesk results examined all project activity, including activity that did not result in changes to the shared outputs of the group. This choice included many user-support discussions on the users mailing list, as well as Trackers that close without changes (invalid bugs or rejected RFEs). These types of project activity are 'discussion-only' and reflect lower levels of time investment than do production work; they may involve more participants.

In this way the results presented in this chapter provide the beginnings of a solid answer to the first research question of this dissertation: work in FLOSS projects, for the large part, is organized to be individual, short and layered. This pattern is coupled with spontaneous support and the deferral of difficult work. The presentation of the research in this chapter surfaced the manner in which the analyst came to these understandings and validated a set of concepts for use in such sense-making.

Yet for a comprehensive answer to the first two research questions of this dissertation two crucial questions remain. How is this type of work able to build complex, interdependent artifacts like software and how does it interact with the motivations of participants (RQ2)? Further, given the frequency of deferred work and the ease of forking or founding competing projects, how are FLOSS projects able to progress sufficiently quickly to satisfy their developers and their user base? To advance answers to these questions the dissertation now turns back to the literature to build a basis for presenting an explanatory model which links the answers to RQ1 and RQ2, and provides a framework to advance answers to RQ3.

Chapter 6

Literature Review III

This chapter introduces literature which is relevant to the development of an explanatory model in Chapter 7. It focuses on literature which provides models of individual motivation and literature which links motivation and production. Finally it identifies risk as a construct that is present in both the motivation and coordination literature, albeit at different levels of analysis. These concepts are crucial to understanding why the collaboration patterns identified in the case studies work and work well.

6.1 Motivation

The study of motivation is vast and varied. The purpose of this review is to identify literature on motivation which enables a better understanding of possible links between motivation and production, particularly as it pertains to understanding feedback and thereby the ability of a project to sustain itself.

A survey of the extensive literature on motivation in the workplace reveals three key theories that appear likely to be linked to organizational structures and interdependence in particular: expectancy-valence theories and self-efficacy theories job design.

Figure 6.1: The steps in expectancy valance theories. Blockages at either step can undermine motivation.

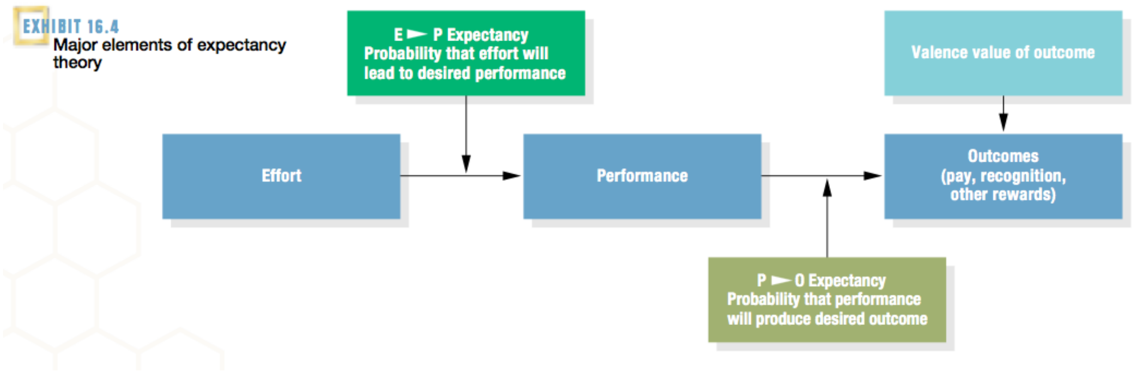


EXHIBIT 16.4
Major elements of expectancy theory
E ► P Expectancy
Probability that effort will lead to desired performance
Valence value of outcome
Effort
Performance
Outcomes (pay, recognition, other rewards)
P ► O Expectancy
Probability that performance will produce desired outcome

Expectancy-Valence Theories

The first relevant set of motivational theories are those called expectancy theories (Vroom, 1964; Porter and Lawler, 1968). These are considered a process theory of motivation:

> *Process theorists view work motivation from a dynamic perspective and look for causal relationships across time and events as they relate to human behavior in the workplace (Steers et al., 2004, p. 381).*

The essence of the expectancy theory of motivation is 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).*

Figure 6.1, from Samson and Daft (2005, p. 534), 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". The second is Performance-Output (P-O) which argues that the individual also calculates the "Probability that effort will lead to desired outcome".

Either expectation could turn out to be unfounded; the individual could be 'blocked' at each stage in the process. For example a programmer might choose a task which is too difficult or time consuming and fail to create working code. Or despite having produced a patch that describes the successful changes they have made to the codebase, that patch might be rejected or reverted by other developers.

Crucially Porter and Lawler (1968) introduce the idea that individuals will learn from such failures, as expressed by Steers et al. (2004, p. 381)

> *if superior performance in the past failed to lead to superior rewards, employee effort may suffer as incentives and the reward system lose credibility in the employee's eyes.*

For the purposes of this dissertation the important message of this motivational theory is that interdependencies could be a source of such blockages and thus demotivating. Failed interdependencies could block, or hinder, the production of a patch or its inclusion in the software. The frequency with which either of these occurs ought to affect the expectations of individuals in the future. This effect could easily spread, since other potential developers may observe such failures and the frustrated emails they usually produce.

It is certainly true that the obverse is also possible; successful collaboration could be particularly motivating because of added enjoyment of teamwork or an experienced responsibility for other's work. Yet the experience in BibDesk suggests that failures might have larger negative effect than successes have a positive effect. Certainly successes are not likely to have the same contagion effects that we suggest failures might have.

Therefore if a successful project needs to attract participants and get their best effort, arranging their organizational structure to avoid the occurrence of demotivating failures seems a useful strategy, if it can be implemented. In Literature Review II (Chapter 4) we identified FLOSS projects as emergent and their task as flexible enough that projects have choices about the levels of interdependency. The evidence in the two studies presented so far suggests that they do indeed choose low levels of interdependence.

Self-efficacy theories

The second major set of motivational theories that appears to be linked to the organizational structure of a FLOSS project are those associated with goal-setting (Locke, 1996; Locke and Latham, 1990), self-determination (Ryan and Deci, 2000) and self-efficacy (Bandura, 1997).

Goal setting theory suggests that establishing challenging, yet achievable, goals increases the motivation of individuals to achieve them. This knowledge has been developed into the increasingly common management practice of working together with employees to establish short and long-term goals, and to prompt performance through 'stretch' or difficult goals (Ambrose and Kulik, 1999). Conversely it is understood that goals are demotivating if they are unachievable or hold little personal challenge.

Self-determination and self-efficacy theories argue that the experience of setting one's own course and believing that one can successfully follow that course is motivating for individuals and potentially groups (Bandura, 1997).

Together this literature suggests that the experience of setting one's own goals and achieving them creates a virtuous, motivational cycle. Interruptions in this cycle are likely to undermine feelings of self-efficacy and control and thus undermine motivations. Both these understandings seem to fit well with the patterns of individual work identified in BibDesk and replicated with Fire and Gaim.

Job Design

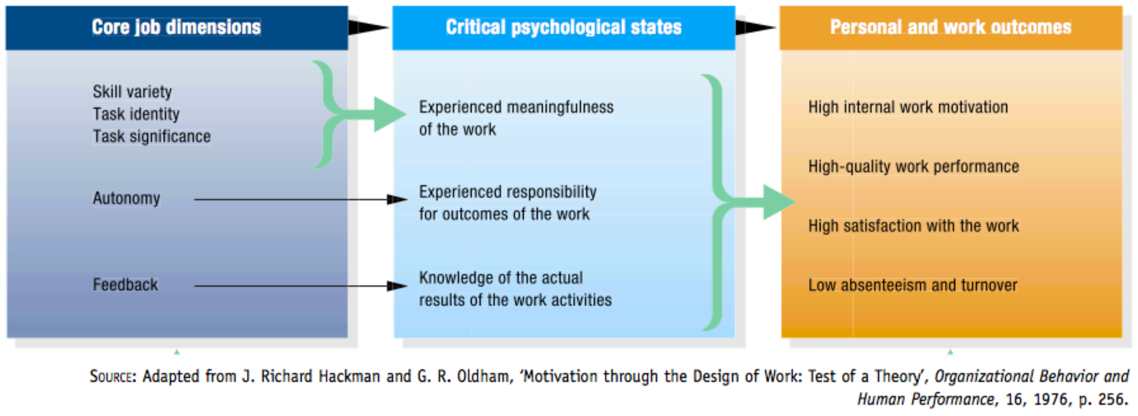
The third set of motivational theories are part of the job design literature (Hackman and Oldham, 1980; Herzberg, 1966), sometimes known as the Job Characteristics Model (JCM). The basic insight of this work is that the nature of the job, and thus the tasks, that people undertake can affect their motivation and their effort during work. Ultimately dissatisfaction leads people to leave their jobs.

The characteristics of jobs considered relevant to motivation are these:

- Skill variety (What skills are needed, simple or complex?)
- Task identity (Does the individual complete the task, or just do a small part?)
- Task significance (Is the task important to the person?)
- Autonomy (How much discretion and freedom does the person have in planning and carrying out the tasks?)
- Feedback (Does the job provide quick feedback to the person about their performance?)

The job design literature identifies the type of individual as important because different job characteristics are important to different types of people. People with high "growth need" tend to be more motivated by these factors (or positive answers to the questions), whereas people with low "growth need" tend to be less motivated by these factors and more by lower level factors such as basic pay, simplicity and free time. Since participants in FLOSS projects are almost always choosing to become engaged above and beyond their regular lives and jobs, it is reasonable to assume that they have "high growth need". Since participants are choosing their tasks (Crowston et al., 2005) it seems that they will adjust the skill variety and task significance factors themselves. In contrast it seems reasonable that the organizational structure of the project will more strongly influence the factors of task identity, autonomy

Figure 6.2: Motivational effects of job characteristics


Core job dimensions
Skill variety
Task identity
Task significance
Autonomy
Feedback
Critical psychological states
Experienced meaningfulness of the work
Experienced responsibility for outcomes of the work
Knowledge of the actual results of the work activities
Personal and work outcomes
High internal work motivation
High-quality work performance
High satisfaction with the work
Low absenteeism and turnover
SOURCE: Adapted from J. Richard Hackman and G. R. Oldham, 'Motivation through the Design of Work: Test of a Theory', Organizational Behavior and Human Performance, 16, 1976, p. 256.

and feedback. Figure 6.2 shows how a common organizational behavior textbook summarizes these factors (Samson and Daft, 2005, p. 543).

6.2 Linking motivation and interdependency

Chapter 4 introduced and examined coordination in production. This section examines the connections between that work and the motivation literature introduced above, arguing that the concept of risk is an appropriate link between the two literatures.

There are a limited set of studies that have directly examined the motivational effects of interdependency. Kiggundu brought interdependency together with Hackman's Job Design (Kiggundu, 1981, 1983). He identified two relevant types of interdependence, one with negative effects on outcomes, including motivation, and one with positive effects. The two types of interdependence identified are *received* task interdependence and *initiated* task interdependence. Thomas (1957) illustrated the difference by describing two people operating an anti-aircraft gun; one firing and the other loading shells. The person firing the gun cannot effectively do so unless they have received a shell from the other person; they have received interdependence. The person passing the shell initiates the process by passing a shell; they have initiated interdependence. The relationship is interdependent because both are working to fire the gun.

Kiggundu argues that received task interdependence is de-motivating because it reduces autonomy, although he does not explore the mechanism in detail. Conversely initiated task interdependence is understood to increase experienced responsibility for the dependent work (and worker) and experienced responsibility is held to improve motivation, commitment and other outcomes, "Members high in initiated interdependence experience a sense of responsibility to maximally facilitate and minimally hinder the task performance of others" (Taggar and Haines, 2006, p. 212).

In his presentation Kiggundu draws heavily on Trist and Bamforth (1951), which is considered a classic socio-technical work and describes interdependence in the "long-wall" coal mining method in English nationalized coal mines. Kiggundu explains their finding of differential levels of outcome variables (motivation, job performance, absenteeism) amongst different roles by arguing that the worse outcomes were found amongst those roles with high received, rather than initiated, interdependence.

Kiggundu's argument has never been consistently integrated into the Job Design literature; however it is more common in the relatively rare cases where the Job Design study is focusing on teamwork (Bachrach et al., 2006; Taggar and Haines, 2006; Humphrey et al., 2007; Van Der Vegt et al., 2000; Bertolotti et al., 2005). Furthermore the majority of this work is committed to the idea that interdependence adheres in the task; it is "rooted at the level of the task … required, rather than … optional" (Kiggundu, 1981, p 501).

Risk as a linking concept

Risk plays an important role in both the literature on interdependency and the literature on motivation, although it is too rarely explicitly examined. The risk of failure is the link between interdependence and performance in both Thompson's and Malone & Crowston's conceptualization of interdependence. The reason organizations ought to seek a fit between the demands of a task and coordination mechanisms is because if they do not they risk the reality of lower performance. A lack of alignment is therefore an organizational risk. This risk, however, is usually conceptualized only at an organizational level. Individuals are obviously affected by organizational performance in the long run, but may be as likely to attribute this to management as to the experience of failed interdependency.

The risk of failure also plays a central role in the expectancy-valence, goal setting and self-efficacy theories of motivation, where experienced failure and perceived risk can both undermine motivation, and thereby performance. This theory, of course, is conceptualized at the individual level.

But in FLOSS projects the division between the individual level and the group level is not so clear cut. Under-motivated individuals cannot be re-motivated by increased salary, and they cannot easily be assigned to work closely with a well-motivated and successful teammate, since the project has little, if any, authority over participants. Projects are therefore quite limited in their ability to choose interdependence strategies that might initiate the positive motivational effects of interdependency discussed above. Furthermore it is clear that attempting interventions like that would undermine the autonomy of participants and could, thereby, cause further motivational drops.

In this way the risk of failure identified in the individual motivation literature is directly related to the risk of failure in the coordination literature. A failure of interdependency is an organizational level failure in that the progress of the project is impeded, but its effect is amplified because it is also an experienced failure for the participants, undermining motivation. In this way the organization of work directly effects the motivations of individual participants and thus undermines the inputs for the next episode of production.

6.3 Interim Conclusions

With this background it is now possible to build an explanatory model which is consistent with the literature and the findings of the dissertation thus far and provides an integrated answer to the first two research questions (organization and its link with motivation). This model will provide the groundwork for advancing an answer to the third, to be undertaken in the Discussion (Chapter 8).

Chapter 7

Formalization: An Explanatory Model

This chapter develops an explanatory model which is consistent with the literature and deeply grounded in the two empirical studies reported so far. The processes described in this model require a set of conditions, found in FLOSS production, if they are to play out. These conditions provide a way to answer the third research question and discuss the adaptability of the type of organization that drives community-based FLOSS projects, a task taken up in the Discussion (Chapter 8).

The model aims to explain three findings so far:

1. Individual work was far more common than Co work,
2. Despite this, projects are able to produce integrated software which is successful, and
3. Participants appear to defer complex work.

In short the chapter takes takes up the most crucial question thrown up by the work so far: Why are FLOSS projects so heavily skewed towards individual work, and how can they succeed like this?

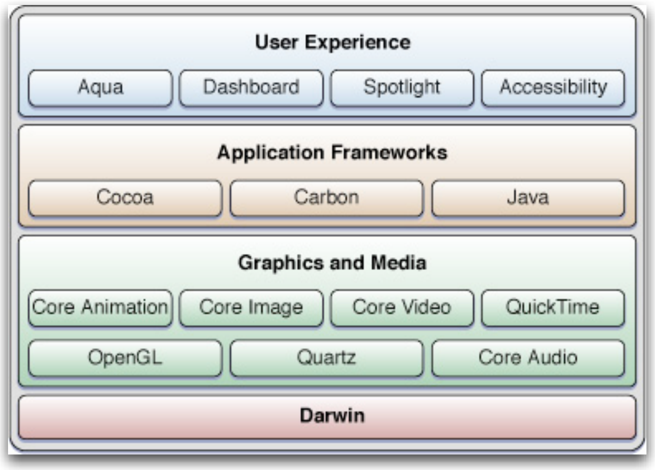
The approach taken in this chapter is to build an analytical model of collaboration, grounded in individual rational behavior. The model restates the problem of collaboration in FLOSS projects using a stylization of software development.

The analysis then considers two solutions to this dilemma. The first is Co work, either contemporaneous or sequential, and it is argued that the high risks of this, under conditions of individual motivation, explain the choice to proceed mostly through individual layered work. The second solution demonstrates a production strategy that is believed to be novel.

Having demonstrated analytical results in a justifiably stylized model of FLOSS production, the chapter works back towards empirical reality, relaxing assumptions and considering their impact on the model. The chapter concludes with a sketch of a dynamic extension which attempts to incorporate the process of recruitment and retention.

The model starts with a very restrictive set of assumptions which intentionally under-describe the reality of FLOSS development. These assumptions actually handicap the "project" in this model, making it less capable than a more empirically true project. Yet, as we will see, the modeled project is still able to achieve development, albeit limited in some ways, through a surprising and important result. The reader is asked to hold off their empirical objections to these assumptions; they are returned to later in the chapter and systematically relaxed. As they are relaxed the model gains in empirical veracity and the project gains more potential ways to advance. However these gains come with risks that, it is suggested, projects must navigate, if they are to be successful.

Figure 7.1: Apple's Architecture Stack for OS X. (From Mac OS X Technology Overview, Chapter 2)



7.1 The Task of Software Development

When programmers speak with each other about software they often talk about a "stack" of software and draw diagrams that look similar to those from Apple (Figure 7.1) and Sun (Figure 7.2). These illustrate a fundamental feature of software: it is built in layers. The lower layers, such as an operating system, provide services to the higher layers, such as a windowing toolkit. Only the very top layers, applications, provide direct utility to users. They are, in fact, called applications because they "apply" the technologies to users' problems. The user doesn't perceive these lower layers directly, they perceive them mediated by higher layers.

This layering is true not only at the very broad stroke of an operating system, but is also true within a particular application. For example the ability for a user to search their email relies on ("applies") at least two lower layers: the ability to match text like words, and the ability to display the search results (email display).

Figure 7.2: Sun's architecture stack for the Java Development Kit 5.0 (From Sun JSE 1.5 Documentation)

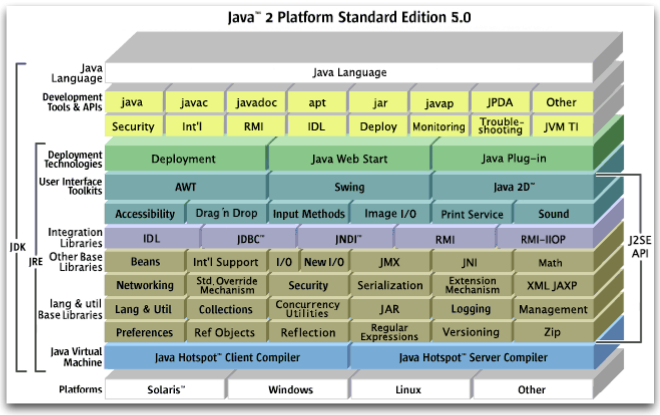
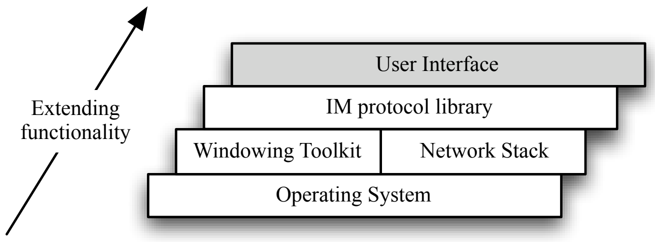


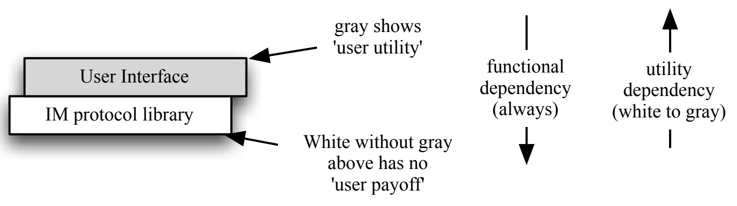
Figure 7.3: Software is layered and additive: higher layers extend its functionality by building on services provided by lower layers.



As these layers extend upwards they extend the functionality for the user, as shown in Figure 7.3

In addition to, or perhaps as a result of, being built from layers, software is additive. That is to say that additional layers can be added without necessarily changing the lower layers. Furthermore, if a layer is no longer needed it can be removed without necessarily affecting layers below it (and since it is not being used there are no layers above it!). This is, of course, merely a way of modeling software development and it does simplify reality. It is quite possible to build software in monolithic chunks, without the layering described here, yet it is equally possible to

Figure 7.4: 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



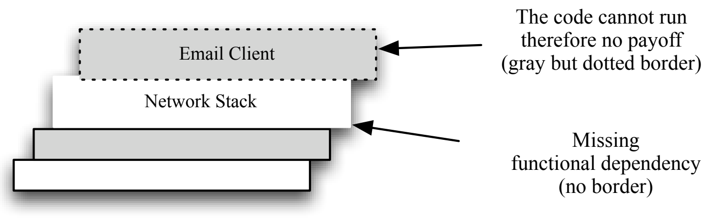
build it through layers and the layer metaphor is very useful in modeling the task that developers face in building FLOSS.

Functional and Utility dependencies

The fact that software can be built in layers gives rise to two important types of dependencies between software layers. Figure 7.4 highlights these relationships. From top to bottom the layers depend on each other functionally; without these 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 7.4 also shows the second type of dependency, called a *utility* dependency. Some layers of software have no direct utility to the user. Rather they depend on higher layers to expose their functionality, thereby releasing their value to a user. In these diagrams layers which have direct utility are colored in gray, while layers which rely on higher layers for their utility are colored white. The Tasks reported for the Fire project in Chapter 5 include many situations like this, such as when the iconv library was incorporated. The iconv library is a very capable character encoding conversion library, but on its own and newly added it offers no utility to

Figure 7.5: 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

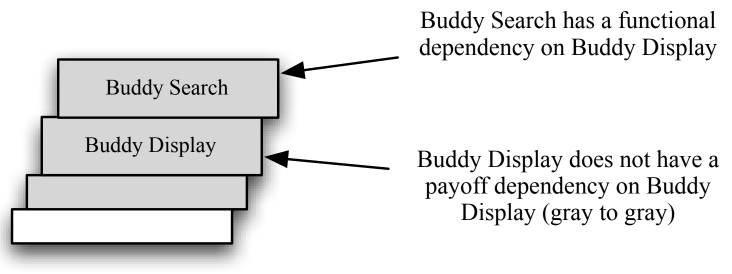


an end user. Rather it forms a new layer on top of which user interface and other integration routines unlock its potential.

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 7.5 shows this situation. In Chapter 3 such a situation was presented, where the author was unable to complete his metadata plans due to the absence of a suitable library.

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 7.6 illustrates this situation with two features of an instant messaging client: buddy display and buddy search, as encountered in the "new search" task in Fire Task 32. Buddy search clearly depends on being able to display buddies, otherwise the results can't be displayed or acted upon. Yet buddy display is already useful, even without the addition of buddy search. 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. The majority of the Tasks considered in Fire and Gaim were like this, with new functionality building on already useful technologies.

Figure 7.6: 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.


Buddy Search has a functional dependency on Buddy Display
Buddy Search
Buddy Display
Buddy Display does not have a payoff dependency on Buddy Display (gray to gray)

With this background we now turn to the model itself, beginning with a brief review of rational choice models, then building up needed assumptions and finally considering how the model plays out through multiple turns.

7.2 Rational Choice Models

The model presented in this chapter is a rational choice model. It attempts to capture essential features of individual's decisions through a simplified, stylized model of human decision making. These models are most familiar from micro-economics where they form the foundation of consumer choice theories, leading to well known predictions such as supply will equal demand if the price of a good is allowed to vary. They are also the foundation of game theory models, as well as the underlying model of action in multi-agent simulation models. Individuals in these models are termed agents, which emphasizes that they are simplifications of humans, albeit analytically useful.

The model presented in this chapter is quite simple, yet is useful for explaining the two empirical findings in the previous chapters: the pre-dominance of individual work and the use of deferral as a production strategy.

The agents in rational choice models are making a choice between alternatives, such as choosing which basket of products to buy, or which investments to make. To make their choice they assess the benefits and the costs of each course of action and are constrained to choose the course of action that yields the greatest net benefit to them. Benefits and Costs do not have to be limited to be entirely selfish, it is possible to argue that a benefit for others is also an outcome which will please the rational agent, and therefore result in an increased benefit to that individual. Likewise there is no need to assume that a benefit for others is a cost to the agent, although that is a common assumption in market competition models (where a sale to a competitor is a sale lost to others).

The basic model is simple: an agent will make a choice to act when that choice yields the greatest difference between benefits and costs (Assumption 1 in Table 7.1). Essentially the agent will act if the Benefits exceed Costs:

$$B > C \tag{7.1}$$

Costs are most usefully understood as "opportunity costs" which are the benefits forgone by not choosing the next most attractive alternative. For example if an individual is choosing between eating an orange or an apple, the cost of eating the orange is the enjoyment/sustenance—the utility—that the apple would have provided and vice versa. This allows rewriting Equation 7.1 so that the agent will act if Benefits exceed the Opportunity Cost of next most beneficial choice.

$$B_{choice} > B_{alternative} \tag{7.2}$$

Of course an agent cannot see the future; they can only make their decision on their expectations, their estimates, of both the benefits and the costs of the action. In short there is a risk that the full benefit will not be realized, while there is no risk that the opportunity cost will be incurred. This allows restating the left-hand side of the decision equation, where $U_{outcome}$ is the Utility derived from the outcome and R

is the risk that the agent's expectations are wrong, and their decision will not lead to the desired outcome.

$$E\left(B_{choice}\right) = U_{outcome} \times (1 - R) \tag{7.3}$$

The Expectancy-Valance theories of motivation outlined in Chapter 6 helps to understand risk here, pointing out that there are in fact two different expectancies in play and a risk at each stage. The first is the expectancy that the actor's effort e will result in a certain performance p, which can be written as a probability: $P(e \rightarrow p)$. The second is that that performance p will yield the expected outcomes o and thus the benefit: $P(p \rightarrow o)$. These probabilities are expressing the probability of the successful outcome, so there is no need to subtract from 1. Now the equation can be further restated, with all components being Expected values:

$$E\left(B_{choice}\right) = E\left(U_{outcome}\right) \times E\left(P(e \rightarrow p)\right) \times E\left(P(p \rightarrow o)\right) \tag{7.4}$$

In actually, of course, there are real values for $P(e \rightarrow p)$ and $P(p \rightarrow o)$, which will determine whether the agent, having committed to the choice, will in fact receive the benefit. The agent doesn't know these real values in advance, but experience may reveal those to the agents over time. In short the agents can learn that their expectations are wrong, and will attempt to adjust them towards the real values. However agents are limited in their ability to predict the future regardless of experience due to its complexity: the agents are therefore said to have be bounded rationality (Simon, 1957). Accordingly we assume that agents are good, but not perfect, judges and are aware of their limitations, thus $E\left(P(e \rightarrow p)\right) \equiv P(e \rightarrow p)$ and allowing $P(e \rightarrow p)$ to be less than 1 (Assumptions 3 & 4 in Table 7.1).

To illustrate this imagine a rational agent trying to decide whether to sell a magic wand prop, or to wave it in the air to magically produce money. $U_{outcome}$ is the value of that money, $P(e \rightarrow p)$ is 1, since the agent presumably can wave a wand if he chooses to (and knows this) but, since our agent doesn't believe in magic, $E(P(p \rightarrow o))$ is zero. A rational agent with these beliefs will always chose to sell the wand. Suppose a second agent does believe wholeheartedly in magic ($E(P(p \rightarrow o)) = 1$), they will choose to wave the wand, forgoing the selling opportunity. Of course, since the real value of $P(p \rightarrow o)$ is 0, the agent will not receive the expected benefit. Eventually the agent will adjust their expectations.

Conversely imagine an agent faced with the choice as to whether to try to make it in professional sports by entering the draft, or alternatively to spend the week of the draft on vacation. The rewards of success ($E(U_{outcome})$) are high, and there is a reasonable expectation that excellent performance at the selection combine will lead to being selected and therefore getting paid ($E(P(p \rightarrow o)) = 1$), since the real value of $P(p \rightarrow o)$ is 1 by rule and is well-known (drafted players receive guaranteed high-paying contracts). The agent, however, is realistic about their skill levels and knows that there is zero chance of being selected $E(P(e \rightarrow p)) = 0$. The agent will chose to go on vacation instead.

7.3 Assumptions

This section details a set of assumptions which are necessary for analytical clarity and justified through literature and the participant observation experience detailed in Chapter 3. These assumptions are stylized but aim to capture important aspects of the FLOSS situation. Unlike many analytical models most assumptions in fact make the task harder for the project. Nonetheless the majority will be relaxed in the discussion below, after the initial analytic point has been made (indicated in Table 7.1 by a ✓). This practice provides clarity to the presentation. The assumptions are summarized in Table 7.1 in the order they are introduced in the text.

Table 7.1: Model Assumptions (✓shows which are later relaxed)

#		Assumption (*Justification*)
		Bounded Rationality
1	✓	Participants will only work for a utility payoff (*Parsimony*)
2	✓	Participants are motivated only by their own use of the software (*Parsi., Lit. and Exp.*)
3		Participants are good, but not perfect, judges of task complexity (*Parsimony and Exp.*)
4		Participants know the limitations of their judgement (expectations approach reality) (*Parsimony and Experience*)
		Other Assumptions
5	✓	All participants have the same set of skills and availability (*Parsimony*)
6	✓	Participants only know their free time for the next turn (*Parsimony and Experience*)
7	✓	There are no exogenous sources of code or solutions (*Parsimony*)
8	✓	Participants can build on existing layers without assistance from authors (*Experience*)
9		Contributions are always shared under an open source license (non-revokable, no royalties, allows derivative works) (*Parsimony, Lit. and Exp.*)

The model considers agents who are potential developers to a FLOSS project. It is assumed that agents have a limited amount of spare time, outside their normal course of life including things such as paid work, family life etc. It is also assumed that these agents regularly use the software outside their spare time, perhaps for work, for study or for managing a family vacation. This assumption is motivated by the experience in BibDesk, where the "real-life" use of the software for academic writing was important, as described in Chapter 3.

Initially it is assumed that agents are only motivated by a desire to improve the software so that it improves the effectiveness of their other activities (Assumption 2 in Table 7.1). That is to say that their motivations for participation are entirely instrumental. (This assumption will be relaxed later). The improvement in effectiveness is therefore the value of $U_{outcome}$ and is known to the agent ($E\left(U_{outcome}\right) = U_{outcome}$).

The regular use of the software allows the agent to see opportunities to improve the software (and thereby the effectiveness of the rest of their activities). These opportunities could be to remove annoyances, such as a clunky interface or data-loss risking bugs, or to extend the capabilities of the software through new features. In terms of the simplification of software development described above both of these outcomes involve the creation of a layer of software, which can either be thought of as re-writing an existing layer to fix a bug or adding a new layer to extend features. For simplicity we assume that a new layer is being added.

Also for simplicity the choice facing the agent is constrained to be binary: they either choose to spend their time attempting to contribute to a FLOSS project or they choose not to. What they do with their spare time otherwise is immaterial, but for the sake of narrative we will model it as though the other choice available to an agent is to meditate. The agents are assumed to be good meditators and meditation is assumed to yield some benefit in the rest of their life (through improved focus or, perhaps, karma). Meditation, therefore, is equivalent to the 'null' choice in investment models, which is usually to place one's money in a bank account accruing low but certain interest. In this model the non-code choice is a choice to meditate, which always produces a known payoff. Choosing to code means that the agent looses the benefits of meditation, so the known payoff of meditation is always the opportunity cost of choosing to code.

The model requires the agents to make two assessments of their expectations. First they have to estimate whether the effort they have available can result in the needed performance, estimating $P(e \rightarrow p)$. They inspect the current codebase and make their best estimate of the amount of time it would take them to build the required layer. Of course this depends on a number of factors. Firstly they have to be able to read and understand the existing codebase. Secondly, they have to estimate their skill levels and thirdly, the complexity of the task. These elements enable them to assess the difficulty of the task and thus the time likely required for the task which they then compare against the time they have available. The agent sets their expectation of $P(e \rightarrow p)$ based on the chance that they will be able to complete the task with the time 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 high, but if they needed to integrate a new spell checking library in that three hours then the estimate of $P(e \rightarrow p)$ would be very low.

Having agents make this assessment requires further assumptions (Assumptions 3, 4 & 5). Initially all agents are assumed to have the same level of skill (this is relaxed below). They are assumed to be good, but not perfect, judges of their skill level. 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. 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 (Assumption 6). Agents will not rely on possible availability in future turns (this is relaxed below).

The second assessment that the agents make is to assess $P(p \rightarrow o)$ which is 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 reflects the fact that the agent has the codebase, a compiler and can compile their layer with the application. In short this is an assumption that the agent is able to successfully integrate their changes into the application. These assumptions and therefore this term will be varied below.

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 (Assumption 7). This rules out outsourcing through payment or the discovery of libraries of code from outside the project. This is an obvious simplification and will be relaxed below.

Further it is assumed that it is possible to build on existing code without needing to speak with the original author(s) (Assumption 8). This doesn't necessarily imply that the code is perfectly readable or perfectly documented, but that the developers are, with time, effort and experimentation, able to read and understand how to use it. The difficulty of doing so is incorporated into the agent's decision about their probability of success. This assumption is justified by the participant observation, but it too will be varied below.

Finally there are two assumptions about contributions, once they are successfully coded. Firstly, it is assumed that the agents all share their contributions (Assumption 9). This is justified by the understanding that once a feature or fix is accepted into the public code-base, others will refrain from removing or undoing it (provided it is functional). The decision to share is rational because this ensures that the layer that the agents builds will continue to function and deliver productivity improvements ($U_{outcome}$ is an assessment of continuing future benefit). This group commitment therefore saves the agent considerable future maintenance costs, much in the same way that corporate IS departments attempt to minimize internal customizations to ERP systems, especially if those might break during upgrades.

Secondly, contributions are shared under an open source license (a license that respects the OSI principles, Assumption 9). This has three important implications: contributions are not revokable; they cannot be withdrawn by the contributor. This does not mean that they cannot be appropriately altered, but that even if a developer were to regret their decision to contribute, they would not be able to remove the layer. This is a result of the use of an open license and considered in more detail in Chapter 8. An open-source license also ensures that contributed code can be relied upon and used by others: derivative works are allowed and no royalties must be paid. It should be noted that this model is not attempting to explain this decision to reveal, rather it is attempting to consider decisions as to the style of work (individual or in groups) undertaken.

Again, these assumptions are made for analytical clarity. The more empirically unrealistic of them are eventually relaxed and their effect on the model investigated, see page 168 or page 176.

7.4 Model Analysis

With these assumptions and background, it is possible to consider an individual agent making a code or meditate decision. We are going to fill out the elements of the equations on page 158 and page 159 with illustrative examples.

First we create some known elements:

- The agent has 10 hours available for the next turn (a week, say)
- The agent knows that meditating for these hours would improve their output in the future by 7 academic papers[1]. Therefore the opportunity cost of coding is 8 units. We now know that the agent will chose to code if they expect that their choice will result in future improvements greater than 8 papers.
- $P(p \rightarrow o)$ is known to be 1, as discussed above.
- The agent consults their preferences and conceives an improvement they would like to make to the application, such as automating a library lookup for online journals. They estimate that this will enable them to improve their output in the future so they will publish 10 more papers this year. Therefore U_{choice} is set to 10.
- The agent then examines the codebase, judging the complexity of the work and comparing that to the time they have available. The agent decides that the task is probably feasible within 10 hours but knows from experience that it may prove more difficult, so that their estimate will be wrong one in five times and their effort will not result in working code after 10 hours of work. Therefore $E\left(P(e \rightarrow p)\right) = \frac{4}{5}$.

The decision equation is then simple:

[1]Of course this is quite unrealistic but simple whole numbers help the illustration because they simplify the math.

$$B = 10 \times \frac{8}{10} \times 1 = 8 \ngtr 7 \tag{7.5}$$

The expected benefit is greater than the opportunity cost ($B > C$), so the agent will choose to code rather than meditate.

In this way the model proceeds in a simple manner. Each participant consults their current set of motivating outcomes and the current codebase and assesses 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 benefits outweigh the costs they begin work.

If they successfully complete the work it is available to all the other developers in the next period, for others to assess their options. Unsatisfied motivations where the developer either chose not to act or did not successfully complete the work will survive through the next turn.

Backwards-only dependencies

The simplest situation is that a participant inspects the codebase and their set of desires and finds a task which is sufficiently motivating, but is also within their abilities given the time known to be available to them. Such a situation is shown in

Figure 7.7: As development proceeds over time through layering, functional dependencies become backwards only dependencies. They depend only on what is already present.

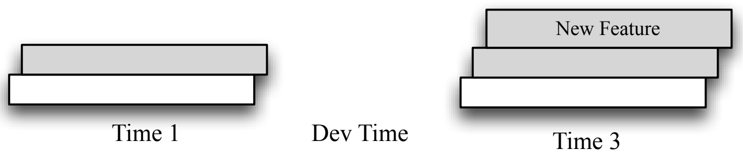


Figure 7.7, where a gray layer (depicting utility to its developer) is layered atop an existing gray layer.

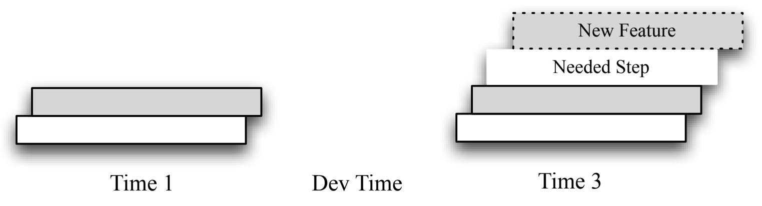
These situations can be called “backwards-only” dependencies: all that is relied on is the current codebase and its permanent availability (as guaranteed in the FLOSS context). If previously contributed layers could be removed then the payoff would not be guaranteed, since if the layers were removed the code would lack its functional dependencies and would be unable to release the utility desired by the developer. The dependency is “backwards” in time, depending on actions already completed, irrevocable and therefore certain, actions of others. All the Solo Tasks found in Fire and Gaim were like this, as was the implementation of the “Container” column in BibDesk, reported on page 42, in Chapter 5.

The Missing Step situation

A second situation faced by a participant would be inspecting the current code base and their desires and seeing 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; if they start they will fail. This can be understood as setting $E\left(P(e \rightarrow p)\right)$ so low that meditation will always provide a higher expected return, regardless of the size of $U_{outcome}$. In our graphic simplification this can be represented by depicting the feature as two separate layers, as in Figure 7.8.

An individual agent under the assumptions above will not work to try to implement both layers, nor is it rational to work on either in isolation. This is because without the white missing step, the payoff of the gray layer is not available (a missing functional dependency). Without the potential to finish a gray layer in the time available the white layer will not be built (since it is missing a utility dependency).

Figure 7.8: A missing functional dependency becomes a forwards and backwards dependency



This is a fundamental dilemma in collaboration; it is in fact just a stylized way of restating a core problem of collective action. We have already encountered these situations in this dissertation. One such situation was related in Chapter 3 where the author's desire to implement automatic metadata reading and writing was blocked by the absence of libraries to edit PDF XML metadata (and the author's lack of ability to implement these alone). Other such situations were likely behind the long-running episodes discussed at the end of Chapter 5. The following sections consider two solutions to this situation: collaboration, which is well known, and deferral, which is novel.

Collaboration as a solution

A natural way to resolve this situation is to have agents communicate and potentially make agreements between themselves. In this way they are free to discover other participants who are also motivated by the New Feature in Figure 7.8. Together they can assess that two developers working together could accomplish the work if one works on Needed Step and the other on New Feature, as depicted in Figure 7.8[2].

[2]These agreements can only be for concurrent work, not sequential. This is because developers are constrained to only rely on availability for the current turn and sequential collaboration would require the first developer to trust that the second developer had availability, but they do not. The point is not crucial, and see below for a relaxation that opens up this possibility.

The capacity to work together opens the way to resolving the dilemma, as depicted in Figure 7.9. The question, then, is why is it so uncommon in FLOSS development? The answer lies in the concept identified in Chapter 6 as linking the literature on interdependency and coordination and motivation: risk.

From the perspective of either developer, this agreement introduces a new source of non-completion risk. This is because any developer's collaborator also has a chance of failure ($P(e \rightarrow p) < 1$) and therefore may not complete their part of the agreement, rendering the payoff unavailable (either because a functional or a payoff dependency is unsatisfied.) From the perspective of the first developer this means that regardless of their ability to complete their part of the task, the reward may not be available. This can be incorporated into the model as a change in the expectation of the second part of the risk term, $E\left(P(p \rightarrow o)\right)$ where regardless of the success in performance there is a risk that the expected outcome will not eventuate. Of course this cuts both ways, reducing the expected value multiplicatively and sharply decreasing the likelihood of agreement and action.

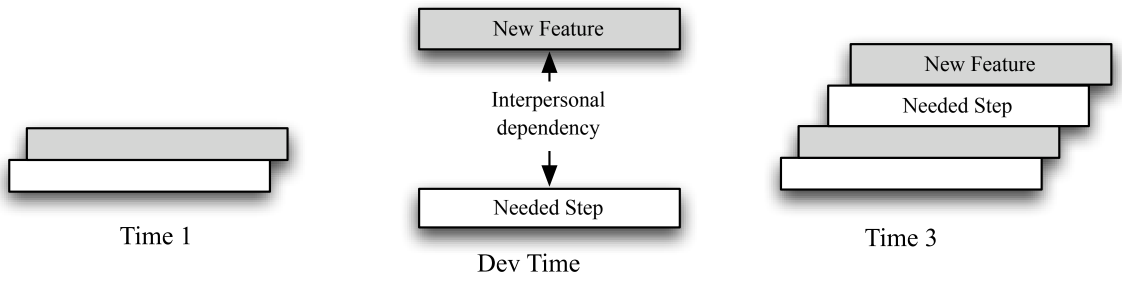
$P(e \rightarrow p)$ is an estimate of the probability that effort in the time available will result in one of the two layers required. If we, as above, set this to 0.8, then the decision equations of both actors look like this:

$$10 \times 0.8 \times 0.8 = 10 \times 0.64 = 6.4$$

This is now less than the benefit of meditation (7), so there will be no agreement and neither participant will choose to code over meditate. Essentially each agent's $P(e \rightarrow p)$ becomes their partner's $P(p \rightarrow o)$.

In addition, concurrent development introduces a new, well known, problem. For two layers to work together they have to suit each other, what Crowston (2003) calls a usability dependency. Until now this has been assumed to be easy (although see 7.5 below) since the developers have only been building on finished code and it is assumed

Figure 7.9: The missing functional dependency can be added if two developers are able to work concurrently, although the risk of non-completion rises due to partner failure and coordination costs



they can understand this without discussion with the authors. This is not true of concurrent work because it is not available for inspection as it is being simultaneously developed. This can be called a usability coordination cost and this extra difficulty can be modeled by increasing the risk of non-completion (for either participant), by decreasing $P(e \rightarrow p)$. Furthermore this is known to participants—and known to affect their partner—so it is therefore transferred to the expectation of the $P(p \rightarrow o)$ term. Representing the increased risk of concurrent programming as θ this makes the collaborative decision condition,

$$U_{outcome} \times (P(e \rightarrow p)_{self} \times \theta) \times (P(e \rightarrow p)_{other} \times \theta) > C \tag{7.6}$$

If we set $\theta = \frac{9}{10}$, saying in effect that concurrent programming difficulty alone causes a failure for the agent 1 time out of ten, and using the values from above, the decision equation becomes:

$$10 \times (0.8 \times 0.9) \times (0.8 \times 0.9) = 10 \times 0.72^2 = 5.1$$

Algebraically it is possible to see that the utility required to make the choice to collaborate a rational one must be substantially larger (where $p = P(e \rightarrow p)$):

$$U \times (p_{self}\theta) \times (p_{other}\theta) > C \tag{7.7}$$

assuming that p and θ are the same for all participants (and all terms are positive) that gives:

$$U(p\theta)^2 > C \tag{7.8}$$

$$U > \frac{C}{(p\theta)^2} \tag{7.9}$$

if C is set to 7 and p set to $\frac{8}{10}$ and θ set to $\frac{9}{10}$, the value of U for which agents will agree to work together can be calculated:

$$U > \frac{7}{(\frac{8}{10} \times \frac{9}{10})^2}$$

$$U > 13.5$$

The required value of U, therefore would be nearly double the opportunity cost of the alternative. The absolute size of this, of course, is dependent on the expectations of the agents regarding failure. The point is that the risks are multiplicative and both $P(e \rightarrow p)$ and θ are less than 1; concurrent work is substantially more risky than individual work.

Interestingly there are no clear-cut examples of this type of agreement in the Fire and Gaim data, even amongst the Co work tasks. The closest is in Fire Task 30, where a developer remarks in a CVS log message, “Jason, you have the set status routine now”. This remark does seem to imply that the author was aware that Jason needed this routine and perhaps wrote it for him (although it seems just as likely that the developer needed it as well). The analysis did not find any evidence of preliminary negotiation, or planning, even for this Task. The other Co work tasks have more than one developer, but little indication that one is explicitly relying on the other's future performance. Of course there may have been out of band negotiations, or perhaps the developers are willing to work without upfront agreement due to past experience or perhaps they are simply happy to have help if and when it arrives, but would work on independently regardless.

Of course the above models agreements as being between 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 further increases risk through extra coordination effort which is now between three or more simultaneously developing components, rather than two.

Finally, until now we have assumed that communication and negotiation for an agreement is costless. It is not, since the two agents must discuss and agree on an outcome and a method of getting there, a process that can be quite costly, especially given the environment of lean communications and unreliable attention that prevails in FLOSS projects. Furthermore these negotiations are separate decisions with uncertain outcomes, which detract from time available to implement simpler, more certain, individual tasks. These costs are identified as transaction costs (Williamson, 1981). They could be modeled in a number of ways, such as introducing a pre-turn negotiation, or by introducing another term akin to the risk of concurrent programming. Either way it would serve to further decrease the expected utility of negotiating a collaborative solution to the Missing Step problem.

In this way collaboration provides a solution to the dilemma but it is one that increases risks in a super-linear manner due to the costs of coordination and transactions. The actual effect of these risks depends on how the participants estimate them, how often non-completion is a problem and any potential costs and risks of negotiation, but in the end always depends on the size of the expected payoff (the desirability of the feature). In some situations, particularly when the task is important or much desired collaboration still may be a rational choice.

At this stage the model has explained why Co work should be expected to be less likely than individual work, as observed in all three case studies. In short, it is more risky. Setting aside, until below, that some Co work was in fact observed, the question arises as to whether any substantial progress is possible for the project under the current model, which speaks to the second question of this chapter: how can complex interdependent software be built when individual work is so dominant?

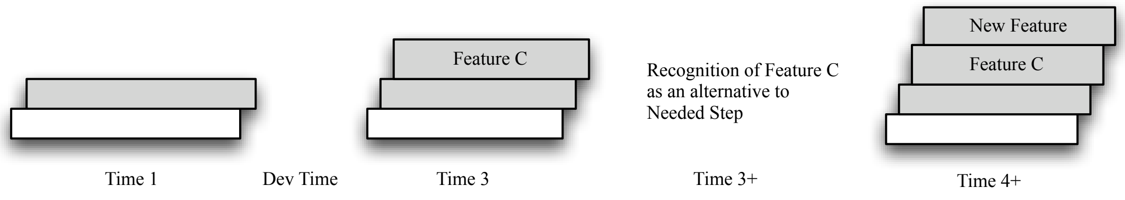
Deferral as a solution

The answer lies in the fact that there is another possibility available without introducing communication and agreements and it has already been seen in the empirical components of this dissertation. This is for the developer to defer the work for the current turn, and to wait and see how the codebase changes over time. It is possible that, through the work of others on the project, the situation will have changed sufficiently that New Feature is now possible with only individual work within one period. In short, from the perspective of a single participant, Needed Step simply appears, and turns the dilemma into a relatively simple Backwards-only dependancy.

What trickery is this? Needed Step is white; it is a layer without a utility payoff and therefore will never be built by participants. It can't come from an exogenous source, because we are 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, but as other work—perhaps just individual backwards-only work—changes the software over time another way to build New Feature may become apparent.

Figure 7.10: 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.



This can occur because, as outlined above, a developer makes their best assessment of the easiest way to change the code to achieve their desired outcomes and bases their calculation of $P(e \rightarrow p)$ on that assessment. The Needed Step/New Feature dilemma is therefore not a hard fact—it is not a "structural requirement" of task as discussed in Literature Review II on page 73—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. This process was observed in all three case studies, although was seemingly more frequent in Fire and BibDesk than Gaim.

In this way, as depicted in Figure 7.10, potentially problematic interpersonal dependancies, requiring trust and communication, can be converted to two backwards-only dependencies, and accomplished through individual work alone.

This can be incorporated into the rational choice model with an acknowledgement that the deferred work may not ever be completed, and if it is it will be delayed and therefore reduce the utility of the feature. Note however that agents are not choosing to defer the work as an alternative to both working and meditating, deferral does not take any time, so the benefits of meditation are still available. Note also that deferral is also not a commitment on the behalf of the agent to undertaking the work in future. In this way one can remodel the decision as between (a) working uncertainly now and likely failing and (b) meditating now and working more easily later. For deferral to be preferred to collaboration, then, the effect of uncertainty and delay must be be lower than the multiplicative combination of collaboration risks shown above.

Of course deferral is indistinguishable from simply choosing not to work in the current turn, in this way it also operates whenever an agent decides that the task is not achievable either individually or with others through negotiating and performing an agreement. It is relevant that deferral eases the implications of choosing not to work; the opportunity to work is not one-time, nor does a single agent's decision not to work affect the ability of other agents to work in the future. From an organizational perspective, the cost of an individual agent choosing not to work is relatively low.

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 probably never happens. Even when it does there is no way to estimate when it might occur successfully. In any case deferring work in the hope it gets easier is far from a wonderful solution: it is at best slow and from an organizational perspective is quite often totally ineffective, especially in circumstances where the time-cost of upfront investment operates (see the Dicussions). In the BibDesk project, the author of this dissertation deferred his work to implement readable metadata in PDFs and that task still hasn't been completed today. Nonetheless, deferral is a novel solution to the missing step dilemma and one which is in-keeping with the motivational and technological environment of FLOSS development.

7.5 Relaxing assumptions

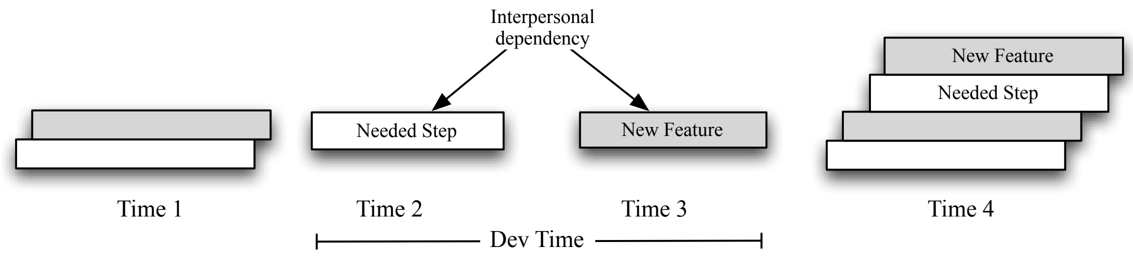
Neither solution outlined above is, on the whole, wonderfully attractive. Happily real-world FLOSS participants are not as constrained as the model has so far assumed. This section fulfills the promises above by relaxing assumptions in sequence. Note that unlike many models, the majority of relaxations make the dilemma less pernicious and easier to overcome. Some assumptions, notably easy integration, do have the opposite effect.

Helping communication

Having developers communicate, even without making agreements, improves the situation for the project. It does this in two ways. The first is to allow others to help participants discover the outcomes which will provide payoffs to them. A common case is for a user to report a bug, or request a feature. If the developer agrees and is therefore personally motivated by this new information, then they have additional outcomes which can motivate work. The assumptions above bind individual agents to generate tasks only from their own use of the software, allowing input from others gives the agents the benefit of other's experiences as well, perhaps allowing an agent to discover a potentially damaging bug and be motivated by a desire to avoid it happening to them. Note that communication can play this role without altering the assumption that agents are only motivated for their own benefit, these benefits are quite apart from seeking to serve others (see below).

A second potential benefit of allowing communication is to allow others, especially other developers, to help conceptualize the causes of a bug, or the steps needed to achieve a desired outcome. In effect this allows the transformation of Needed Step situations towards something closer to a backwards-only dependency. This is similar to the discussion of deferral on page 173 but rather than playing out over time as the

Figure 7.11: A sequential dependency. This helps solve the dilemma, but introduces both inter-personal dependency and delay as the work is not available for use in until time 4



codebase changes the situated cognition of the developer, the re-conceptualization can occur in a single turn and the work can proceed at a quicker pace. As Eric Raymond quotes Linus Torvald, "Somebody finds the problem and somebody else understands it. And I'll go on record as saying that finding it is the bigger challenge." (Raymond, 1998).

Assumptions about availability

There are two assumptions in the model about time. The first is that participants cannot predict their availability beyond the current turn. This can be relaxed in two ways. The first is to allow participants to assess their availability for individual future periods. This combines with agreements, as described above, to make it is possible for two (or more) participants to plan to work together sequentially to achieve a desired outcome, with one participant having current availability implementing the Needed Step and the second participant, having availability at $t + 1$, implementing the New Feature. This is shown in Figure 7.11.

Clearly this shares some of the features of concurrent development outlined above. For example a failure by either is a failure for both, so the $P(e \rightarrow p)$ of each is the

$P(p \rightarrow o)$ for the other. Similarly the transaction costs of negotiating an agreement are still present. Sequential development, however, does not have the additional risks of both components shifting simultaneously; the second developer would just react to the state of the first layer as with backwards-only dependent work. However because sequential development delivers the utility of the layer later, there should be some discounting factor. In this way a choice between concurrent and sequential development is a trade off between concurrent coordination costs and delay.

A second assumption about availability could allow participants to better understand their general future availability. Indeed participants could expect to have a constant availability (such as expecting 10 hours availability each week for the foreseeable future). This would enable participants to engage in their own sequential development, working on tasks which take longer than a turn, provided their utility was high enough and the delay discount was not too large. Of course this would still multiply the risk of failure. Further, participants could be inaccurate in understanding their future availability and find that real life interferes and delays the work sufficiently that its utility is undermined and work eventually abandoned.

A distribution of individual capability (skills and time)

The second assumption to relax is that the contributors have the same set of skills and productivity. Rather these aspects can be modeled as a distribution of both skills and productivity amongst the developers. This can be done without altering either the time-availability or time-horizon for payoff.

The immediate impact of this is to allow the existence of developers for whom the implementation of Needed Step and New Feature (from Figure 7.8) is converted to a single step. This might be for two reasons. Firstly, the developer might have experience of Needed Step through another program, and is therefore able to implement it quickly. Secondly, the developer is able, due to their superior skills and productivity, to complete both of the steps in the free time they know they have available to them. Graphically this either merges the white and gray boxes, or it enables some participants to build two boxes within one turn.

Relaxing this assumption leads to a more realistic model where some tasks are hard for some developers while being easy for others. Indeed it is often said of open source projects that they have "code gods" who are an order of magnitude more productive, and whose work makes leaps on top of which other developers can build smaller, but useful, contributions.[3]. Changing this assumption would complicate the calculations of risk in collaboration, since one developer might have less chance of failure than the other.

Just as there is a range of tasks with different difficulties and skill requirements, there can be a range of developers with different skill and productivity profiles. In this way the project can speed through some Needed Step dilemmas, provided a developer can be found for whom the task is comparably easy *and* provides sufficient utility to that person.

This 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 these thorny issues. This is true even if contributors cannot contribute large amounts of time.

Exogenous change

In the real world, 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

[3]González-Barahona and Robles-Martínez (2003) confirm the often observed skew of productivity in FLOSS projects, but also argue that different people or groups step forward at different times, in the majority of successful projects, questioning whether so-called "code gods" are in fact particular individuals

possibility of code movement between some sets of projects, in general developers have a great deal of software at their disposal. This is especially true of libraries, which are packaged and documented for re-use. As discussed above in section 7.1 these libraries 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 the course of their single turn, in a manner analogous to re-conceptualizing the problem with assistance. The developer, therefore, is assessing not just the current codebase but also the codebase of other projects and out to the whole ecosystem of (compatible) FLOSS code.

Even with the large amount of code available this turns out to not be as great as it seems at first. Many important components are not available as libraries and discovering appropriate code is far from effortless. Such search costs reduce the likelihood of projects adapting exogenous code. There is little doubt that such search effort, as well as the "not invented here" syndrome (perhaps linked to learning, see below), limits the progress that could be achieved through code reuse.

Nonetheless it is clear from the archival study of Fire and Gaim that new underlying libraries provide substantial leaps forward. In the period studied Fire integrated a new library (iconv) and updated three (libmsn, libyahoo2 and libfaim). A developer may integrate a library for a specific Task, but the other features of the library are now more easily available to other developers as they estimate the work needed to implement the task they are motivated by. Thus libraries are "bundles" of potential features which can spark new rounds of creativity amongst developers.

Alternative motivations

The set of assumptions above assumes that all developers are only ever motivated by the use they personally can make of the software. This assumption is fundamental to some of the results of the model outlined above, but is also clearly not realistic.

Studies have, of course, found a range of motivations amongst contributors in open source beyond the product itself, as discussed in the first Literature Review (on page 15). In particular three additional sources of motivations have been cited: intrinsic motivations, reputation and helping others. Intrinsic motivations can refer to a whole set, but include learning, fun and ideologies.

Intrinsic Motivations

Once intrinsic motivations are allowed in the model the situation can vary quite significantly because they change the types of payoffs permitted. Intrinsic motivations allow participants to implement steps which were previously ruled out because they did not provide immediate functional payoffs. This has the helpful effect of making it possible to turn a 'white' box (a step with functionality but no immediate instrumental utility) into a 'gray' box where the payoff is, for example, the learning the developer anticipates during development. This allows intrinsically 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. Of course their contribution could enable another to build New Feature more easily (but this won't have been the original intention or motivation).

Furthermore intrinsic motivations mean that even ultimately unsuccessful work—which the BibDesk participant observation found a lot of—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 can 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 a very powerful motivator.

Formally this can be incorporated in the model by adding (not multiplying) a new term to the equation on page 159 on the benefit side reflecting the expected utility of the experience $E\left(U_{experience}\right)$. This term is unaffected by the risk of failure (or any of the multiplicative risks of collaboration), thereby making work more likely to be preferred.

$$E\left(B_{choice}\right) = E\left(U_{experience}\right) + E\left(U_{outcome}\right) \times E\left(P(e \rightarrow p)\right) \times E\left(P(p \rightarrow o)\right) \quad (7.10)$$

Successful work done for intrinsic 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.

Reputation

Reputation can function in similar ways, motivating developers to take on tasks without certainty of completion or immediate payoff in the software. As long as the project is careful with assigning 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 might be tasks such as managing infrastructure. Reputation, as a type of external reward, could even motivate unpleasant work, just as financial rewards do. Note however that reputation would take time to build, so is unlikely to be operative in the very early stages of a project.

This relaxation allows projects to proceed through blockages without resorting to work with direct interpersonal interdependencies, still working only through layered individual work. Yet reputation seems likely to be important for working in concurrent or sequential modes as well, since reputation can be modeled 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 for concurrent or sequential work.

Helping others

In addition to these important effects, allowing payoffs outside the software itself also sheds more light on a crucial behavior: that of working to solve other people's problems, perhaps out of a sense of generalized obligation to pay-forward the benefits the project has given you. Above we considered the situation where users could provide help conceptualizing or noticing tasks which developers then took on as their own, but if developers are motivated by alternative payoffs this creates the possibility of acting to fix a bug or add a feature even if the developer himself doesn't see the issue or won't use the feature. The specific alternative motive here is not important: it could be learning, reputation, generalized reciprocity, or even an enjoyment in seeing one's software widely used.

Motivations of this type are important since they allow developers to be motivated by increasing user numbers. Greater user numbers mean a larger pool of potentially motivating reports or requests, as well as a larger potential for assistance in characterizing and thus re-conceptualizing the effort required to achieve the desired outcome.

It is certainly the case that developers do sometimes implement things that they themselves have argued are useless, and have done so at the request of users. In one task in Fire (Fire Task 27) a developer noted that he implemented "Invisible (even though it is pretty stupid)"[4].) The converse is perhaps more common, however. The core developer at present in BibDesk is quite outspoken in refusing requests that he assess (with the silent support of the rest of the developers) as being beyond

[4]http://sourceforge.net/project/shownotes.php?release_id=127461group_id33772#release_note

the scope of the project (such as customizable keys), protecting against code and preference bloat.

Each of the assumptions relaxed above have had positive effects on the work of the project, reducing the frequency or impact of the Needed Step dilemma and increasing the potential for both individual and, to some extent, concurrent or sequential work with interpersonal dependencies. Yet it is vital to also revisit the assumptions which eased the task of the developers.

Difficult Integration

In the preceding discussion it is assumed that adding a layer of work to an existing codebase is a relatively easy process. This is the result of the explicit assumption that the existing code is well-organized and documented and the developers are excellent code readers. Furthermore it ignores the effects of more than one developer working independently on the codebase during any one turn. Such concurrent development may shift the ground underneath the developer and when they finish their work they may easily find that the work of others has changed the codebase in ways that render their layer incompatible. Such situations are common and known as "conflicts" (Fogel, 1999).

Conflicts may be trivial to resolve, or they may be complex. Trivial conflicts might be caused by something as simple as adding a function and therefore moving other functions within a file, or renaming functions or variables used elsewhere to improve the readability of code. Complex conflicts involve incompatible logic between two contributions and solving them requires delving into the logic of the conflicting contribution. A concrete example of a complex conflict is a "race condition" where one component is paused ("blocked") waiting an anticipated behavior from another, but the other is also blocked, itself waiting for the behavior of the first component.

One way in which conflicts can be modeled is to introduce a special period at the end of each turn which enables developers to inspect their work in the context of the other work in the period. The contributors add their changes to the codebase in a random order, but must resolve any conflicts their addition creates. If the conflicts are trivial then they can make simple changes and have their contribution added and be the base for the next participant's integration work. However if the conflict is complex and cannot be resolved then their development efforts fail for that turn. However the developer is well placed to use their following turn to perform the complex integration work, with updated knowledge of the codebase. Of course their payoff would be reduced through delay and their layer is unavailable to others for an additional turn.

It is worth noting that the source code repository software used by FLOSS projects specifically assists in discovering conflicts based on the syntax of the code, and some projects also use test suites which can assist in discovering semantic conflicts. Fogel (1999) contains an excellent summary in the FLOSS context. The role of this technology will be further considered in the Discussion.

Thus far we have assumed that all participants start and end their work at the same time, but in reality work is of different lengths and starts at different times, which also helps to mitigate the impact of difficult integration. The model can be altered to allow this by offsetting participants' turns and either varying their length (perhaps as a result of varying skills or challenges) or allowing participants to see the intermediate results of other's work. Both operate by changing the codebase that the participant sees when they begin their task. Offsetting and varying lengths of tasks allows participants to begin their work after a short task by another, which delays payoff but eliminates conflicts from concurrent work. Since the turns are offset the delay effect might not be substantial, since the developer might be working in a different area of the code or might simply not be available for work. Allowing participants to see the intermediate results of each other's work can ease integration because they can spot potential conflicts early on and alter their techniques to reduce the chance of complex integration conflicts. Essentially this allows participants to start their turns during other's turns, or to act early to reduce the effort required for resolving conflicts.

It is worth noting that both of these techniques can work without developers communicating, other than to watch each other's code changes.

Difficult integration but allowing communication

If acknowledging difficult integration is combined with having developers communicate there are more opportunities for easing its effects, but they come with potential costs. For example, in the simplest case a developer is simply trying to understand the completed, unchanging, code on which they are building so that they can assess the effort and process required to achieve a desired outcome. If they can seek the advice of the knowledgeable authors, such as the original author, this may be simpler. However this merely pushes the analytical question back a level: why would the original authors expend energy to answer their questions?

If only instrumental motivations are allowed, it might be possible that the original developer is motivated to 'defend' their code against the possible introduction of bugs during integration. Similarly the knowledgeable developer may also be motivated by the contribution that the other developer is trying to make, which frames this situation as a simple agreement (to attempt to answer questions, should there be any). Finally, the knowledgeable developer might be aware of the effect described on page 173, whereby changes in the codebase have the potential to make tasks they wish to accomplish more possible.

However if one allows alternative motivations it is much simpler to see other developers "helping out", since they might be motivated by reputation (to show their code to be well written) or learning (to see problems others have with their code and thereby write better code, as Lakhani and von Hippel (2003) found) or simply motivated by a desire for the project to succeed, even if they don't care for the outcome themselves.

If one allows communication and allows either the offsetting of tasks of various lengths or access to intermediate code then communication could also assist in heading off potentially complex conflicts, rather than resolving them later. However communication like this is not without cost; it is additional work over and above the development effort and the effect on payoffs is uncertain, since a developer would have to act without being certain that a complex conflict will emerge. They would also have to act without knowing whether their action will necessarily help the other developer, or whether it will simply produce more time-consuming questions.

In reality both watching each other's code and talking about one's own code can be quite difficult, especially because the course of development is often not well planned and while the final product of development might be comprehensible and logical, that does not imply that the work was done in simple pre-planned steps. In some cases access to intermediate work could cause premature adjustment and thereby increase integration effort, and thus the possibility of failure or delay. Similarly in some cases communicating in words about code can be quite hard, especially if the code is unfamiliar to the developers. Such communication could easily require substantial effort and itself become a source of misunderstandings.

7.6 Aligning work with motivations

Thus far the work in this chapter has advanced a rational choice model which goes some way to explaining both the dominance of individual work (since collaboration is risky) and the way in which deferral works. The model is, however, relatively static.

The remainder of the chapter sketches how the motivational findings of the BibDesk case study and the literature reviewed in the previous chapter permit the injection of dynamism into the model.

The model above provides some important and surprising results. However it understates some important interactions between the organization of work and the motivations of participants. The basic model assumes that a participant's motivations are unaffected by their past experience of work in the project, since they are solely driven by the outcomes of their work. Motivation is therefore completely exogenous, generated only by the use of the software. Yet it is clear from the experience in BibDesk and the theories of motivation discussed in Chapter 6 that motivation is likely to be affected by the experience of work.

This can be modeled by introducing an endogenous effect on motivations and allowing this to shift agents' expectation of the probability of success as well as the actual probability of success. This is justified because successful completion of tasks provide a confidence boost while failure undermines confidence. Confidence is important when one considers that participants are making their decisions based on their expectations. Their expectations about their own self-efficacy, their confidence, therefore affect whether or not they will attempt work at all. If they do begin work and encounter difficulties, confidence may assist them in pushing forward to complete the work or even prompt more effort to seek help in the project's public forums. Furthermore efforts to engage other members of the community may increase confidence, while unsuccessful efforts may decrease confidence.

The size of the confidence effects, as with all the parameters in the model, are difficult to estimate, but are related to the size of the effort invested. Thus attempting and failing will have larger effects if the attempt involved a great deal of effort. This implies that failing at a development task will have a larger effect than if one's questions are not answered, simply because development tasks require more effort than composing an email.

It also offers the potential to acknowledge attribution effects, whereby failure that is perceived as the result of others is considered more demotivating than one's own failure, as suggested by Kiggundu (1981). A similar thing could be achieved if participants are allowed to learn from the work done during their own failures, but not through the failures of others. This has the effect of increasing the risks of interpersonal dependency, making individual work more attractive. However one must also acknowledge a potentially reinforcing payoff from the experience of successful Co work, prompting more Co work in the future.

Adding endogenous motivation to the model gives it dynamic feedback effects, whereby successful completion of tasks makes future effort more likely and failure makes future effort less likely. Projects with participants that meet their goals will move forward more quickly and therefore have a codebase that supports more desired outcomes. Such a codebase will also change more frequently and is therefore more likely to increase the possibility that deferral will allow agents to re-conceptualize their seemingly hard problem into an easier one that can be undertaken as individual, less risky, work. Conversely a project where the developers are not achieving their goals suffers from the reverse effect, making deferral less attractive and, over time, reducing the pool of development effort.

Over time this makes projects, through their participants, more likely to stick to things that work, and less likely to undertake more risky actions, modeling the evolution of preferences regarding interdependency observed by Wageman (1995). This is true whether one allows individual participants to learn, or models preferences for types of work as an unchanging individual characteristic, whereby those with preferences that are not rewarded simply cease participating in the project. Of course empirical reality argues for a combination of these factors.

Recruitment effects

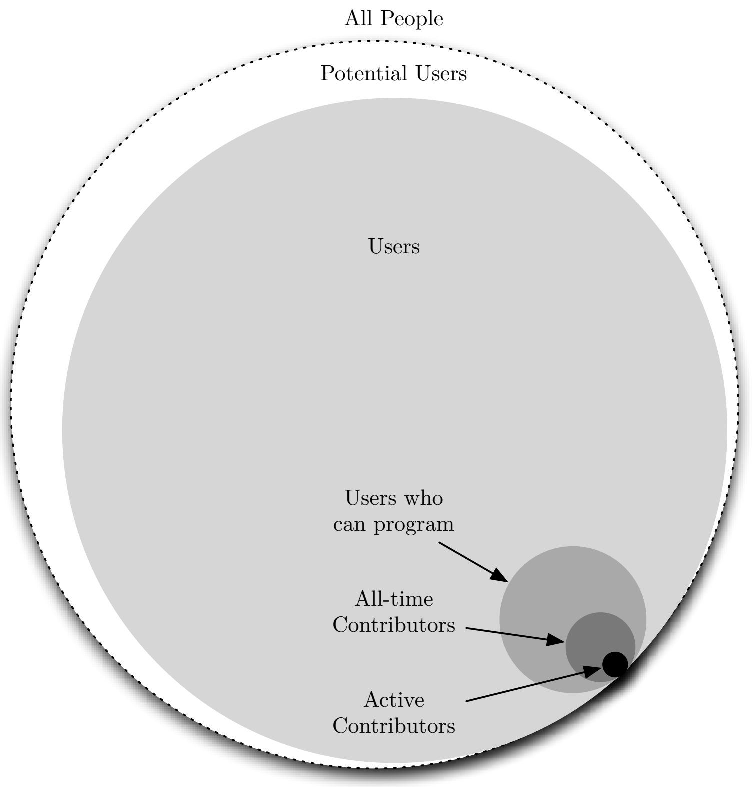
Little has been said about how new developers come to be interested in the project; participants were modeled as a fixed pool attached to projects. Yet this is far from empirical reality, where individuals are free to pick and choose the projects they pay attention to (if any at all). This section discusses two elements related to this: competition between projects and recruitment from a userbase. Both these aspects engender positive feedback effects, but, as is normal in dynamic systems, their success also brings potentially limiting risks.

Projects do not exist on their own, especially in popular areas. The stories of competing FLOSS projects are legion: from Perl/Python/Ruby to emacs vs vi. Similar projects can be seen as competing venues for developer's work. If work is more likely to be successful in a different project, and migration is a possibility, then developers are likely to migrate to such projects. Of course migration has its own costs, since it implies the need to 'get up to speed' with other codebases or social practices. Nonetheless if work within a project is failing, or slow development is undermining the ability for deferral to restructure problems, then migration is a possibility and projects which are good environments for work will move ahead, increasing the bandwagon effects discussed above. Fire eventually closed entirely when the remaining two developers decided that their efforts would be more productive in the faster moving AMSN project.[5]

A similar effect can be obtained if one acknowledges use of the software as a source of development motivations and allows a separate pool of participants who use the software but do not develop for the project. As the usefulness of the project's output increases more users will be attracted. As discussed above it is possible that user numbers might motivate some developers, but these effects are quite possibly

[5]http://sourceforge.net/mailarchive/message.php?msg_name=E6234655-FCC6-46E0-8B81-DF805F3EA861%40tamu.edu

Figure 7.12: A figure showing illustrative ratios between users, potential contributors (users who can code), those that actually do contribute at any time and those who contribute frequently. The outer dotted line indicates a scale change, clearly the universe of all people is massive compared to most potential users, and actual users.



counter-balanced by the emergence of a user-support load. This argument is similar to the model of progressive involvement, based on an evolution of motivations, made for Wikipedia by Crowston and Fagnot (2008).

If one allows users to be recruited as developers, even at very small rates, then the effects are similar to attracting new developers from other projects. Provided they engage in work that is successful the application will increase in utility, the codebase will offer more opportunities for layered enhancement and change to allow deferral to operate effectively. It is likely that such a process would see user numbers growing exponentially (given a large enough potential audience), with contributor numbers growing at a slower, linear, rate.

Both of these recruitment processes seem likely to lead to bandwagon effects, but unrestrained growth is unlikely, since growth also has well-known costs. Rapid growth in the codebase will tend to undermine the ability of individual developers to maintain their understanding of it, reducing their ability to quickly and correctly estimate the work needed to undertake a new task. Furthermore rapid changes tend to receive less testing and therefore have more bugs, reducing the usefulness of the software. This is only increased by the "feature bloat" and "preference bloat" which rapid parallel development can lead to. Maintaining an understandable and reliable codebase under these circumstances requires continuing maintenance work, such as re-factoring, which does not directly increase the usefulness of the software and is therefore not motivating to many participants. Further, such work requires a comprehensive understanding of the growing codebase and the political skills to hold eager developers in check, while not discouraging them from all participation. In this way the bandwagon effects of allowing endogenous motivations will, under some circumstances, be restrained.

This analysis provides some explanation of the bandwagon effects seen in FLOSS projects, whereby the initially successful accelerate while the initially unsuccessful do not and eventually die (e.g. Conklin, 2004). It also offers scope for projects to be initially successful but to reverse course (e.g. Schweik and English, 2007). This is true in a number of ways. For example a project might face competition which provides a more productive environment for developers or, emboldened by initial success, the project might increase its interpersonal dependencies. While this may increase the project's speed in the short term, if anything undermines the availability of partners (such as the natural expansion and contraction of available free time) then collaboration failures, which are especially penalizing in terms of motivation, could quickly undermine the initially successful project. This is discussed further in the Discussion (Chapter 8).

7.7 Limitations

Among the limitations of this model is that the model of software development is relatively simplistic. For example it focuses only on the writing of software, not on its design, testing and periodic re-factoring. This is because these aspects are de-emphasized in FLOSS development, but their absence does question the general applicability of the model.

Further the empirical work grounding the model has been focused on end-user oriented software, where a growth in immediately useable features is most clear. The model may not adequately extend to the development of intentionally infrastructural software, where the work is less likely to be immediately useful to participants. Certainly in commercial software development there is an understanding that preparation work sometimes pays off well down the track, and it is not too hard to imagine participants making similar trade-offs in areas such as library or operating system development.

7.8 Conclusions

This model reveals a surprising affordance of software development which allows projects to overcome a restatement of the collaboration problem by converting it into a set of sequential technology-mediated backwards-only dependencies. Such individual work has the important characteristic that it is always in-synch with the empirically dominant participant motivations and the capability of available communication technologies. The effectiveness of deferral also allows projects an alternative to potentially risky and complex inter-personal dependencies. In the Introduction

(Part I) it was stated that this dissertation is about the "something else" that is invoked in discussions of FLOSS, something novel. This is a large part of that something.

Progress through deferral of complex work is vital and interesting but also limited. It seems destined to be very slow and often fail completely. Relaxing the assumptions of the model brings the picture closer to empirical reality and gives projects more options to overcome blockages in their development, especially once learning is an allowed motivation. The characteristics of the individual decision making environment enable the project, at an organizational level of analysis, to move ahead more quickly, further improving the environment for productive deferral. Nonetheless growth is not unrestrained, especially if there is little motivation to conduct regular maintenance on the codebase.

This chapter has presented a model which is grounded in and explains the core findings of the empirical studies in this dissertation: the dominance of individual work and the use of deferral as a production strategy. Together the empirical findings and the model provide an answer to RQ1 and RQ2. Furthermore because this type of work can only occur under a certain set of conditions it provides a way to understand the circumstances to which the organizational innovations of FLOSS development might successfully be adapted. The next chapter considers how these conditions relate to other work environments, and therefore answers RQ3.

Part III

Conclusions

Chapter 8

Discussion

This chapter draws together answers to the three research questions of the dissertation and reviews their limitations. It particularly focuses on the question of the adaptability of FLOSS organizing, showing how the model developed in the previous chapter provides a way to analyze where else this type of organization might be successful. This is demonstrated by considering three non-FLOSS organizational contexts: Wikipedia, Open Hardware and political advocacy.

As stated in the Introduction, the three research questions of this dissertation are:

1. How is successful FLOSS production organized?
2. How does this organization interact with the motivations of developers?
3. What are the implications for the adaptation of the FLOSS model of organization in other environments?

This chapter will examine each in turn, summarizing the answers to RQ1 and RQ2 already presented and providing an extended discussion of how these answers provide a framework to answer RQ3.

8.1 RQ1: How is successful FLOSS production organized?

This dissertation makes the argument that successful FLOSS production is organized primarily as layered individual work. Such work is chosen and conducted by an individual participant and conducted over short periods of time. It is not conducted alone, but rather "in company" in the sense that it is visible to others in the project and they sometimes assist in unplanned ways. This visibility is closely associated with the technologies that FLOSS projects use, particularly the integrative technology of CVS and the manner in which email can support both asynchronous and near-synchronous communication. It allows others to be involved as opportunities presents themselves, without creating dependencies which might delay the central participant.

Not all work, however, is individual. A small amount of tasks are conducted in a way that requires participants to trust each other and to reciprocally alter their own activities to match those of others. This dissertation has argued that this type of work is rare because it is risky and particularly so in a volunteer environment.

Furthermore FLOSS projects are sometimes able to advance simply by deferring complex work. This is efficacious because an ever changing codebase provides opportunities to re-conceptualize the work needed to implement a feature or fix a bug, sometimes making a seemingly complex task much easier. Libraries and other code produced outside the project can be particularly helpful here.

These findings are supported by evidence from participant observation and the pilot archival study of BibDesk. They were replicated and focused in an archival study of two similar projects: Fire and Gaim. The model developed in Chapter 7 clearly predicts the dominance of individual work found in the two empirical studies.

Of course there are limitations to this result. Firstly the result is limited to projects which are within the scope of this dissertation, as outlined in the Introduction. This means that there is little reason to expect this result to hold in a project simply because that project has released software under an open source license. If the project is, for example, like MySQL in that everyone who works on it is under contract to and paid by a single organization one ought not expect the results to hold.

The result is also limited, even within its scope, by only having studied in detail individual release periods of projects. In BibDesk this was not so problematic because the pilot study confirmed understandings built over four years in the field. In Fire and Gaim, however, this is not the case, The periods studied may have been special in some way. For example not all of the major contributors to the projects were active during these periods. There is no reason to see these periods as special, but the result would be stronger if other periods also showed similar results.

Finally the result is limited, as discussed in detail on page 132, because the identification of Tasks and their classification is the result of qualitative coding undertaken by a single researcher without the benefit of reliability testing that coding by multiple coders would provide. This is mitigated somewhat by the deep involvement of the single coder, who needed to draw on his experience both in FLOSS projects and in software programming to create the Tasks and assign the codes.

There are (at least) two reasonable objections to the reasoning for this finding, both linked to the question of a dependency structure being encoded in the task. The first objection is simply to say that the reason that the work was conducted individually was because the tasks themselves had no interdependence, they existed in a parallel fashion without affecting each other. This allows one to argue that this is the nature of the applications studied and therefore the task structure assumed this shape. The argument here is essentially over the direction of the causation. This dissertation contends that a combination of the motivational context and communications infrastructure interacts with participant's experience to promote undertaking work in an individual way. It is a structurational, emergent argument. Unlike many situations studied in traditional coordination literature, such as restaurants and even commercial software development, FLOSS participants are relatively free to choose their level of interdependency due to the flexibility of the software and the lack of organizational oversight, investment and control, and they choose to pursue the work with minimal interdependence in the conduct of the work (the artifact is, of course, interdependent). Thus this dissertation sides with those studies of coordination and interdependency, such as (Wageman and Gordon, 2005; Faraj and Xiao, 2006), which argue for an emergent view of interdependency in work.

The second similar objection is that individual work may indeed have been preferred for motivational and infrastructure reasons, but that this is a result of achieving modularity in the software, as discussed in Chapter 4. Recall that the basic argument is that effective parallel collaboration is achieved through pre-planned design such that sections of the code are separated from each other and communicate only through well defined interfaces (Baldwin and Clark, 2001; Langlois, 2002; von Hippel, 1990; Parnas et al., 1981). Yet, as has frequently been observed amongst those promoting agile software techniques, this requires understanding the problem that the application will solve in advance (Beck et al., 2001; Warstaa and Abrahamsson, 2003). In sum the argument, and Conway's law, is about designed software. In FLOSS development the assumption that one is designing software to a well specified set of requirements is clearly not true; rather the application develops as users (or clients) come to understand their problems better through situated interaction with rounds of prototypes.

A formal analysis of the modularity characteristics of the software studied in this dissertation was not undertaken, but there was certainly no evidence of strong norms of code ownership or documentation of interfaces, indeed there was very little planning of any kind. Further it was clear that many tasks, by different actors, touched the same core files, but without evidence of the structure at a functional level this is only indicative. Certainly, the applications were all originally developed by a single individual, which would tend to promote non-modular designs (Conway, 1968), only later moving—in an unplanned way—into group development.

In short, this dissertation argues that the work was incremental at a feature level, but that this is distinct from structural modularity in the codebase. Incremental features, driven by dynamic individual motivations, and not modularity as it is traditionally conceived, explains the dominance of individual work. When everything the participants in a project do is subject to revision, re-conceptualization and extension, it seems "always premature" to solidify a design that was appropriate for a particular set of problems for a particular set of participants, at a particular time. Nonetheless the relationship between modularity in the codebase and incrementalism in development merits further research.

8.2 RQ2: How does this organization interact with the motivations of developers?

This dissertation provides two types of answers to this research question. The first is the experiential, introspective examination of motivation as a participant in BibDesk. The author experienced motivation primarily from the application itself and secondarily from learning. The study identified day-to-day use as crucial to the motivation process: annoyances while otherwise working combined with the easy availability of the source code lead to quick investigations of possible fixes. Furthermore the process of a fix, particularly successful ones, often threw up other tasks at exactly the time when confidence was high. Conversely any interruption to this process was demotivating, particularly when work was blocked by circumstances outside the participant's control.

The second type of answer returned to the literature and examined theories of motivation, particularly expectancy-valence theories and theories that brought together motivation and production, such as the theory of job design. These match the experience in BibDesk suggesting that tasks which are achievable, complete (in the sense of task identity), self-chosen, executed with autonomy are more motivating and therefore more common. Working 'in company' can create a sense of responsibility to others, but failures outside individual's control are demotivating.

These two answers were combined in the explanatory model which argued that work can proceed surprisingly well just through these relatively individual, short self-chosen tasks. It was argued that this is true even in circumstances that would traditionally have required reciprocal collaboration because deferral in an active project allows participants to re-conceptualize their task as individual work at a later date.

Furthermore it was argued that these individual successes increase confidence, encouraging participants to take on more tasks. Conversely, it was argued that any failures of others in collaboration are especially demotivating since they do not even result in learning from failure and are outside the control of individuals.

Recruitment was addressed only through an extension to the model: a growing project is more useful, attracting more users, some small portion of whom have the skills needed to be developers. Their day to day use of the application will generate annoyances or desires and, if the source is available and understandable, some small portion of these will be recruited. This seems likely to produce bandwagon effects because more development brings more layers and features as well as ensuring that the project is active enough for deferral to be a productive strategy.

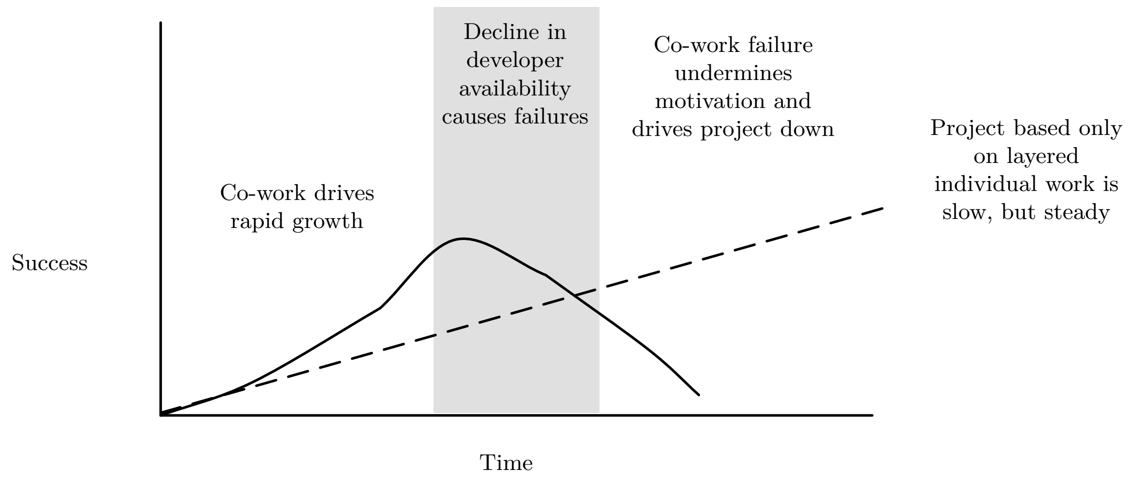
One implication of these answers for FLOSS success is that layered individual work may prove to be more sustainable than organization based on Co work, particularly if it is a default "fall back" for the participants. As discussed in Chapter 5 in the periods studied Fire and Gaim were comparably successful. In early 2004, though, their paths diverged, with Gaim moving from strength to strength and Fire gradually becoming an inactive project, effectively ending in early 2006. While a complete study of the failure of Fire would require further in-depth analysis, when the developers announced "Fire's End" they pointed primarily to a lack of developers able and motivated to maintain and extend the project. The two remaining developers moved together to a similar project called AMSN.[1]

In this context it is interesting, although far from conclusive, that Fire also displayed more Co work, more distinct actors per task and generally longer Tasks than Gaim. In the period studied this did not seem to be a problem for Fire; there was no evidence of failures in Co work. However if a project establishes a norm of Co work early in its life but then the developers find that they have less time available to the project down the track—for whatever reason—then the project may have trouble adjusting back to individual work and therefore experience continuing failures of Co working. Figure 8.1 shows a highly stylized depiction of such an effect.

If the theorizing about motivation is correct then while successful Co work may drive a project in its initial phases but any failures of Co work (for whatever reason) would quickly drag the project down, since failed reliance on others is especially demotivating. Conversely a project that proceeded only through layered individual work might progress more slowly, but ultimately prove to be more robust and successful in the long term. The norms of layered individual work might persist even when the participants, increasingly comfortable with each other, chose to work together for some tasks. If that collaboration broke down the project could fall back on successes in layered individual work to carry it through. Note that the patterns of Fire and Gaim do not exactly match this highly stylized simplified model; it is by no means the whole story.

[1]http://sourceforge.net/mailarchive/message.php?msg_name=E6234655-FCC6-46E0-8B81-DF805F3EA861%40tamu.edu

Figure 8.1: A highly stylized comparison between a project based on Co work (solid line) and one based on layered individual work (dashed line). The initial success of the Co work heavy project is undermined by compounding Co work failures and the slower but more sustainable layered individual work proves more successful in the long run.



8.3 RQ3: Implications for adaptation

Part of the excitement about FLOSS is that it succeeds in surprising circumstances and many hope to adapt its socio-technical infrastructure in different environments and for different problems. Understanding the extent to which this is possible requires a way to grasp which features of context and process are similar enough between contexts to support the model of organizing advanced in Chapter 7. Many of these features were introduced in that chapter but this section draws them together and reiterates their roles. This enables an examination of three alternative contexts, pointing out the important similarities and differences and thereby demonstrating the manner in which this dissertation answers RQ3.

The features of context that are crucial to the operation of the model in Chapter 7 can be organized according to the IPO framework introduced in Chapter 2, as shown in Table 8.1.

Table 8.1: Conditions for the FLOSS model of organizing

	Input
1	Ultra-low upfront investment
2	Individually motivating work
3	Past work is available, non-revokable and non-exhaustable
	Output
5	Instantiation is ultra-low cost and near-instant
6	Distribution is ultra-low cost and near-instant
	Process
7	Task can be approached in layers
8	Task is rewindable
9	Work and communications are observable
10	Communications support temporal mode switching

Ultra-low upfront investment

Upfront investment, such as raising capital, sets a time-clock running and creates a hierarchy of control. The over-riding principle of organization in this circumstance is efficiency: the time-cost of money ensures that the gathered resources must be used to accomplish tasks in the minimum time available, creating an external push for deadlines and deliverables. This undermines the ability of the project to defer difficult work, forcing it to seek to overcome "Needed Step" situations through pre-planning and inter-personal dependency. This immediately suggests that deferral will be difficult to employ productively in a commercial software development context.

Upfront investment of capital also creates a hierarchy of control. The originator of the capital requires a return on investment, either through money (for profit) or through impact (non-profit). This creates a situation of collective responsibility amongst the participants which is quite apart from their personal motivations for participation. This responsibility is met by yielding control, at least at a broad strategic level, to the funder. Even at a tactical day-to-day level the delegation of control to managers is a central feature of organization under such circumstances.

Individually motivating work

The absence of upfront investment also removes a standard motivating mechanism: the exchange of money for time and effort. Without this the project is constrained to organization based on individual motivations. As discussed in Chapter 7 these can be of many types but they must be fundamentally individual, as opposed to collective, at least initially. Of course they can be both extrinsic (e.g. for the software) or intrinsic (e.g. for learning or fun).

In environments lacking both individual motivations one would need to rely on collective motivations, such as altruism, ideology or reciprocation, which are indeed sometimes found (see Chapter 2). However, each of these is fundamentally linked to effect. If the project is already succeeding it is possible that such motivations might provide additional resources, but collective motivations seem unlikely to start or sustain the organizing reported in this dissertation on their own. Collective motivations seem likely to be additive, layering on top of already effective projects driven by individual motivations.

Past work is available, non-revokable and non-exhaustible

A fundamental requirement of the model of organizing advanced in this dissertation is that new work builds on old work. The "backwards-only" dependency, shown as gray-on-gray in the diagrams of Chapter 7, relies on three crucial features of past work.

The first is that past work is available in a form that makes it understandable and permits further building. In FLOSS projects this is usually accomplished by providing anonymous read access to the project's source code repository, but historically has also been accomplished by providing regular tarballs or patch sets. Very simply, if the work is not available there can be no organization of the type described in this dissertation.

Similarly past work must be non-revokable. This is fundamental to a "backwards-only" reliability. If the availability of the work can be revoked then any participant is relying on all past contributors to continue to make their contribution available. This would mean that all work has a future dependency on non-revocation by contributors, turning all work into a sequential dependency where the second step is non-revocation, making all payoffs contingent. Other massive voluntary projects have suffered from revocation problems. For example the CDDB project accepted public contributions of music album listings in exchange for free access to the combined database. The controller of the database revoked the free access, starting a company called Gracenote which charged companies like Apple for access. Needless to say the public contribution component of the project stopped immediately.

Interestingly open source licenses, and the ten point open source definition[2], say nothing specific about non-revocation. This is because all open source licenses are already copyright licenses, and not contracts, and even a normal closed source license has the attribute of non-revocability. Once the source-code is released under a particular license it can be re-released under a new license by the owner, but the owner cannot rescind the rights already granted by the earlier release. Contributions to FLOSS projects "keep on giving", no matter how many people build on or use them. Contributors do not have to continue to supply the resource, as they would if they were contributing bandwidth, for example. This ensures that non-revocability is passive, not an active decision.

Non-exhaustibility is vital to the growth and continued success of this model of collaboration and sets up a fundamental condition: for this model of organizing to work contributions must be non-exhaustible. This is a characteristic of an informationalized product; it can be reproduced at near zero cost and its reproduction does not affect future reproductions—it is infinitely sharable.

[2]http://opensource.org/docs/osd

Instantiation Costs

The source code of a project on its own is of very little use to those seeking to use the application for its intended effects. This is true whether the user is a developer or simply a passive downloader. It is only when the source code is instantiated as an application that the payoff utility of the contributions can be realized.

In the case of software this instantiation is a matter of compiling the source code into a binary. A binary is a set of machine language instructions able to be executed on a compatible computer. Thus the payoff for software development requires both a compiler and a compatible computer. Since compilers are also available as FLOSS and computers are plentiful and widespread, instantiation has an ultra-low marginal cost: simply the effort of a single individual to compile the software and a double-click by the user to start the application. Similarly instantiation is ultra-quick: compilation takes seconds or minutes and executing the software is similarly quick. This is the foundation of "release early and release often" (a well known exhortation in FLOSS circles).

Stepping back from software it is easy to see that not all products have these characteristics. For example a house, a submarine or an airplane all have informationalized representations in their blueprints and production manuals. Yet they are expensive and relatively slow to instantiate, requiring man-years of work and using materials which are exhaustible. No one would live in a house "released-early" and, while producers might like to "release often", the costs of instantiation limit their ability to do so.

Distribution Costs

It is also vital that the product be ultra-cheap and ultra-fast to distribute. The common nomenclature for distribution is revealing: FLOSS software is "released", while commercial software is "shipped". Releasing is a near-zero effort and cost process; one simply lets go. Shipping is a slow and costly process, replete with dependencies.

Consider the situation that prevailed in the software industry before the internet. Software was equally cheap and fast to instantiate, but it had to be distributed on physical media. This process is expensive and takes time. CDs must be stamped, packaged and transported to customers. Once the software is with a customer any errors must be patched by a new release which can only be done by physical reproduction and transportation. These costs create natural deadlines, reducing the possibility of development through deferral and requiring upfront investment that generates a push to pack features into each new release in order to extract payoff in increased sales.

By contrast internet distribution leverages the spare capacity of existing end-user and producer investments in bandwidth and availability. Very few people obtain their internet connection for the purpose of obtaining FLOSS; it is the “chunky” nature of the distribution pipelines that creates the excess resources that ensure that the marginal cost of distributing, or obtaining, software is closer to releasing a helium balloon rather than shipping a CD.

Again not all products are cheap to distribute; it is an attribute of fully informationalized products. And sometimes even informationalized products have costs of distribution. For example a documentary film can contain copyrighted and restricted images in the background of a shot (Hughes et al., 2008; Lessig, 2002). As soon as the film-maker wishes to distribute this to customers they face a choice: either they make their product liable to legal action (and therefore revocable) or they must clear the images for use, definitely expending search costs and in all likelihood being required to pay a royalty for each item distributed. This gives insight into why the first item of the open source definition requires that all open source licenses must rule out royalties.

Together the attributes of ultra-low cost, ultra-quick instantiation and distribution work together with the availability, non-revocability and non-exhaustibility of past work to provide a crucial platform for the model of organizing described in this dissertation. They are closely linked to informationalized products, enhanced by the open source definition. They form a set of criteria for the product which is useful for understanding the applicability and trade-offs inherent in trying to apply the FLOSS model of organization in other domains.

Layerable

For individual layered work to be possible the production environment must be supportive. The product must be able to be built through addition and the layers must be relatively easy to integrate and frequently have independent payoffs for very small additions ("stackable incentives" or perhaps "incremental incentives").

Some work is not like this. For example long-form communicative work, such as in reports or novels, requires attention effort from the receiver and therefore it is vital to deliver them as parsimoniously as possible. Such productions are also most valuable when they are consistent and logical throughout. One cannot fix a sculpture by changing a preference item, one should not ask the recipient of a political pamphlet to spend three days reading it.

The requirement that layers frequently have independent and incremental payoffs is also significant. An airplane certainly can be built in layers: first the fuselage, then the engines, then the wings. But until it is complete none of these steps provide any utility payoff. Similarly releasing a political document or a novel too early may ruin its effect and reputation and it may easily be too late for corrections.

It should be noted that FLOSS is not immune from these effects, in fact not acknowledging them might be a cause of project failure. For example while software is generally layerable if its initial release does not provide sufficient utility for regular use, the positive feedback of annoyance recruitment cannot occur. Of course the product should not be perfect, either. Eric Raymond captures this idea in his advice that a new project release only when its software provides “plausible promise” (Raymond, 1998), which others have characterized as best done with a “Cathedral before the Bazaar” (Senyard and Michlmayr, 2004). If a project is too ambitious early on and attempts to solve too many problems for its first release, it may exhaust the initial goodwill of its participants and never achieve the sustainability of a culture of layered individual work, as seems to have happened with the Chandler project (Rosenberg, 2007).

The second aspect of layerable work is that the layers must be relatively easy to integrate. If they are not, then achieving a payoff through adding a layer is harder because there are actually two tasks, producing the layer and adding it to the project. In Chapter 7 this ease was initially assumed but then relaxed to discuss how source control repositories like CVS make this task easier for FLOSS development by making conflicts more obvious and their resolution more individual.

This point is best understood by considering integration work as just another type of task that ought to have the same characteristics as regular tasks. That is to say it should be relatively clear what is required for the integration and it should be able to be done in an individual fashion, perhaps assisted by opportunistic, not planned, supporting work.

Again not all work has this characteristic and not even all work under an open source license has this characteristic. For example, Apple Computer, working in commercial secrecy, built the Safari web browser on top of the open source web rendering library called khtml. The library was under the LGPL license so Apple was within their legal rights to work in secret and only had to release their modifications once Safari was finished and distributed. When Apple released Safari they did indeed release their modifications. Yet the khtml project was quite displeased. This was because the modifications were too large and had branched from khtml too long ago for them to be easily integrated. As one of the khtml authors wrote:

> *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. Many of us wanted to even sign NDA's [sic] with Apple to at least get access to the history of their internal vcs and be able to be merging the changes incrementally, the way they can right now. Nothing came out of it.*[3]

Such "code bombs" are an indication of the importance of small layers as contributions; they maintain the understandability of the code and are far more easily integrated.

Rewindable

This previous quotation also points to another important characteristic of work suitable for successfully adapting the model of FLOSS organizing advanced in this dissertation. The more rewindable the work, the better. Rewindability has two effects. First it increases the understandability of the codebase because one can see the dynamics of the code, rather than just a static picture. Secondly, it provides a route for recovering from failures which reduces the impact of conflicts between unplanned individual contributions.

This is a socio-technical characteristic of work. It is supported, but not guaranteed by the "sequence of patches" view of work facilitated by CVS and improved by later source code control systems. Developers speak of "atomic commits" which has both a technical and a social meaning. Its consensus technical meaning is simply that"a set

[3]http://www.kdedevelopers.org/node/1001

of distinct changes is applied as a single operation"[4]. In source code the set of changes is across different files, but these changes work together to produce a single impact on the runtime application. CVS actually is quite poor at this (it manages revisions at a file level) and it was a specific reason for the development of its replacement, Subversion (which advances the revision count for all files in the module if even a single file is changed).

However atomic commits are also a social practice, as demonstrated by the following quotation from a FLOSS developer's blog,

> *One thing that I try very hard to be diligent about is the 'atomic commit'. That is to say, when you commit, you're committing exactly one change. It might be across a bunch of files, but it's one change. The reason is that if you later realize it was a bad choice for whatever reason, you need only do a reverse merge (merge -rN:N-1) to undo it. If you don't have an atomic commit, and you only want to undo part of what you committed, then you're stuck with doing that merge, but then undoing part of the merge, which gets really dicey if you've got some mods in a file that you want, and some mods that you don't. Ick.*[5]

While the technical system can enforce a syntactical atomicity, only intentional practice can ensure semantic atomicity and thus real rewindability.

Not all work has this characteristic. As with questions about interdependence this is sometimes a fundamental structural characteristic of work but probably more often a question of the socio-technical production practices and therefore amenable to re-design, usually through increased informationalization. For example consider two people working on a marble sculpture. They have to adjust to each other's work; if one makes a change—lops off the head, say—the other cannot rewind and begin from before that action. Even the two working together cannot accomplish that, short of

[4]http://en.wikipedia.org/wiki/Atomic_commit

[5]http://barneyb.com/barneyblog/2006/01/27/atomic-commits-to-version-control/

some system of nano-scale engineering to rebuild the marble. Much service work has this characteristic, from hair cuts to financial advice: once it is done, it stays done.

Conversely some work can be re-designed to be more rewindable, usually by keeping a closer audit trail and informationalizing the work. Scientific workflows, such as Taverna, offer this potential to scientists (Howison et al., 2008). Consider wanting to change a graphic in a paper during a revision for a journal. If the graphic is the result of a rewindable and re-playable workflow that holds all the working from data, analysis and graphing, then the researcher has a much easier task, particularly compared to having had a student produce the graphic manually on their personal laptop and then graduate, taking everything with them.

Rewindability reduces the need to plan well in advance. Rather than 'measure twice, cut once' with rewindable work one can cut at will when the motivation strikes, and measure later to recover if the work was problematic.

Features of communications infrastructure

The model presented in Chapter 7 suggests that individual work is opportunistically enhanced by participation of others, especially in roles other than the core code producer for a software layer. The key to this type of assistance is that it is opportunistic, which is to say that it is unplanned for and not relied on. It is "nice if you can get it" but does not create a risky dependency. This only works if others can see public records of the developer's actions, both programming and discussion. Such real-time records signal that someone is currently paying attention to the project and is therefore available for an opportunistic interaction. This type of transparency and observability is part of what Kellogg et al. (2006) call "trading zones".

Some forms of communication do not meet this criteria, such as daily digests or fully moderated venues (moderation introduces delay). Other venues are closed in nature, either as a policy decision (closed membership) or as a practical matter (because potential contributors don't have access to the medium of communication, as with ham radio) or because the medium is one-to-one as opposed to broadcast (like the telephone or private email). Still others do not have a push attribute, such as forums, and rely on an out of band notification system, such as an RSS feed.

An additional affordance of communications able to support opportunistic collaboration is that they be capable of temporal flexibility and threading. Together these features allow the 'upgrading' of conversations from asynchronous memo-like transactions to near-synchronous conversational interchange. Crucially the temporal flexibility of the medium allows the conversation to move smoothly between these two forms, without requiring a change of venue and archiving system or pre-planned shifts. Email lists have this curious affordance while many other capable media, like telephones, do not because they have limited archiving capability. Threading is important because it allows participants to conduct multiple, overlapping conversations without extra effort to distinguish between topics (as is required in IRC chats, for example).

8.4 Potential for adaptation

The conditions outlined above, in conjunction with the model outlined in Chapter 7 allow us to consider challenges to adapting the FLOSS model of organization in other environments. This section considers three, from most likely to least likely: Wikipedia, Open Hardware and documents prepared during political advocacy.

Wikipedia

The reasons for the success of Wikipedia seem reasonably likely to be similar to those suggested for FLOSS in this dissertation. Indeed Wikipedia was explicitly modeled on FLOSS collaboration. Wiki projects have little to no upfront investment and the vast majority of participants are volunteer individuals working without pay or other formal external motivation. The motivations of participants are not as well researched as for FLOSS (but see Kuznetsov, 2006; Nov, 2007; Schroer and Hertel, 2006), but it is generally held that people are working on the basis of internal motivations.

Crucially, the GNU Free Documentation License[6] currently used throughout Wikipedia ensures that past contributions are available and non-revocable, stating explicitly (point 9) that,

> *parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.*

It is plausible that participants are motivated by annoyance while reading an article and quickly assess the effort required to have it match their expectations, contributing in short interactions that build up overtime and are available to future participants (Crowston and Fagnot, 2008). These contributions are immediately available—even faster than in FLOSS—because they aren't moderated or otherwise delayed. In this way the payoff is immediate available, whatever motivation is driving it.

The work is clearly layerable through individual opportunistic contributions, even more so than FLOSS; pages often have many separate contributors, even if one or two people contributed the bulk of the work. Empirical work on these patterns of contribution is emerging. For example, Kittur and Kraut (2008) examined the concentration of editing on a page, finding a Gini co-efficient of 0.25 and a median number of all-time editors per page of 11. They also found that the more concentrated pages tended to be of higher quality and that the core group of editors tended to make larger contributions (as well as being more frequent) (Kittur et al., 2007). Similarly

[6]http://en.wikipedia.org/wiki/Wikipedia:
Text_of_the_GNU_Free_Documentation_License

Ortega et al. (2008) found that fewer than 10% of the editors were responsible for over 90% of the English contributions (and that this typically varied between 80%–85% in other languages).

These figures are not directly analogous to the work reported in this dissertation, rather they are analogous to earlier similar findings regarding skews in contributions per file in FLOSS. The task-level analysis presented here would require examining work in a more contextual fashion, finding a middle-ground unit of analysis somewhere between all-time contributors to a page and individual edits. Aspects of Wikipedia, such as on-going improvement projects and category wide metadata systems may exhibit higher levels of Co work.

The availability of full edit history and easy reversion means that it is rewindable, something that is more important and far more common than in FLOSS development. The process of editing and submission allows participants to integrate their elements *in situ*, although the emergence of a hierarchy of editors indicates that integration of new content is a separate task in the Wikipedia world. Judging what is meant by acceptable integration is a far more qualitative task when a compiler is not immediately indicating all syntactical and some semantic conflicts.

The communication structure of the collaboration is simpler than that of FLOSS, being almost exclusively undertaken through the Talk pages that accompany each page. Longer term projects for improving Wikipedia, such as Featured Content or cross-page topical improvement projects are conducted in open ways that facilitate casual contribution, while still providing overall structure.

The clear similarities should not obscure the differences, however. For example the Wikipedia Foundation was formed prior to the existence of the material, rather than afterwards. Furthermore, while the content is in principle non-exhaustible, in practice Wikipedia is an enormous consumer of bandwidth, far beyond even the whole Sourceforge site. This is because unlike a downloadable application, Wikipedia is essentially instantiated and distributed for each individual use of the shared product, and though each individual view is cheap, in aggregate this is very expensive. Future research should carefully assess where to draw analogous boundaries between projects and supporting infrastructure providers, like Sourceforge.

Open Hardware

The Open Hardware movement is a second online collaboration which has explicitly attempted to adapt the FLOSS model of organization. This has been less successful than that of Wikipedia. The aim of the Open Hardware movement is to radically lower the cost of hardware and to radically increase the speed of innovation. Examples include projects like the Simputer[7] and OScar[8] (Open Source Car).

The primary impediment to repeating the success of the FLOSS model of organization 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 this reason Open Hardware has proceeded in two ways. The first is focusing on the design stage, sharing schematics and other documents. The second is to take advantage of the increasing informationalization of hardware.

Focusing on the design stage is relatively simple because designs are information artifacts, very similar to software. Circuit board designs can be tested *in silica* through simulators which work very much like compilers. However the entire enterprise has a clear unsatisfied utility dependency: in order to deliver the use payoff the informational presentation is not sufficient, the actual hardware must be built.

The second method is to take advantage of increasingly flexible and reconfigurable hardware, such as field programable gate arrays (FPGAs) and other programable logic

[7]http://www.simputer.org

[8]http://www.theoscarproject.org

devices. This provides the opportunity to update hardware "in the field" (at the end-user's premises), providing the opportunity to lower instantiation and distribution costs. Nonetheless the sheer physicality of hardware, such as a car, means that its production and distribution are always going to be more costly than software alone. Burns and Howison (2001) argued that this feature protects hardware from impacts of informationalization, such as the Napster effect on music. Yet it also restricts the usefulness of the FLOSS mode of production, short of the development of effective fabricator technologies

Some point to nano-technology and 3D printers to radically informationalize hardware, which might not be that far away[9]. If the promise of these technologies is realized then the potential for organizing the production of hardware based on techniques like layered individual work and deferral will improve.

Policy advocacy

There have also been calls for bringing FLOSS organization to voluntary political activity, such as groups seeking to contribute to various "calls for comment" in government decision making (e.g. Cogburn et al., 2005). Some aspects of this work seem promising for a FLOSS model of contribution: the work is informationalized (producing an electronic written document) and can be built in layers. It is quite plausible that the participants are individually motivated and little upfront investment is required.

Yet two aspects limit the applicability of the organizational techniques described in this dissertation. The first is that the processes these efforts are focused upon have inherent deadlines because contributions have to be finished and submitted at a certain time set by governments. The second is that the overall intention of the activity, and likely its motivation, is dependent on the document having an effect

[9]http://reprap.org

on political decision making and the real world. The path to such effects is not impossible, many have identified epistemic communities as effective in policy making (e.g. Haas et al., 1977). Yet even when successful the payoff is necessarily delayed and individual layers do not have their own payoffs. In these ways the work is more similar to attempting concurrent development of a single layer of software, and the risks of collaboration are not defrayed.

In summary these three explorations demonstrate a central point of this dissertation: the socio-technical organizational structure at play in community-based open source projects is quite specific and will not easily adapt to other environments. The Introduction pointed to the prevailing optimism that FLOSS can teach us a great deal about distributed work. The work in this dissertation suggests that this process will need to be far more nuanced than expected. For example, it seems unlikely that commercial software development companies will seek to adopt the practice of deferral (since it is likely to lead to missed deadlines), yet this practice is fundamental to achieving the dominance of layered individual work that lines up so closely with the individual motivations of participants. Unpicking the reinforcing bundle of institutions, characteristics, practices, technologies and organizational attributes seems likely to require a great deal of innovation to be successful.

8.5 Conclusion

This Discussion chapter has demonstrated that the dissertation has provided answers to all three of its research questions. The empirical results and the explanatory model provide answers to the first two research questions and describe what is different or unique in FLOSS organizing; its "something else". The conditions necessary for the model to operate provide a framework to consider efforts to adapt the model for other types of work and in other environments. This framework was used to examine three real examples of attempted FLOSS adaptation and is therefore an answer to the third research question.

Chapter 9

Conclusion

9.1 Contributions

Until this dissertation there has not been a model of FLOSS organization that draws together the motivations of participants, the technologies of collaboration and the experience and organization of production. This dissertation has presented such a model, shown that it is closely grounded in the empirical data and demonstrated its usefulness in assessing where FLOSS organization might successfully be adapted.

This dissertation makes a contribution to Information Systems because it is a socio-technical theory where the IT artifact (Orlikowski and Iacono, 2001; Benbasat and Zmud, 2003) plays a double role. Firstly, as the subject of production, the flexibility of the IT artifact plays a crucial role because it allows layering and deferral to work. Software is also easily and cheaply instantiated and this, combined with the low cost and high speed of internet distribution, is crucial to the motivational feedback cycles thought to be important to developer motivation. Finally, the specific affordances of technologies of collaboration such as CVS and email play a central role in the theory. For example the ability to identify and help resolve conflicts that CVS offers its users are vital to realizing the additive and layered potential of software.

Similarly the temporal flexibility of email allows participants to opportunistically support each other in their largely individual work.

A contribution is also made to Organizational Science because a new model of organizing is described and analyzed. In particular this model of organizing examines a phenomenon where the task is not structurally determining the appropriate way to organize, but rather organization is emergent, shaped by the resource context and extremely flexible technologies of production and collaboration. In addition the description of "strategic deferral" as an effective component of organization that converts a situation which would usually require risky reciprocal person-to-person dependency into two relatively straightforward person-object dependencies is believed to be novel. In addition a contribution is made to the tradition of studies of interdependency described in Chapter 4 by studying the type of interdependency that emerges in a volunteer, distributed environment with a flexible task.

Finally this dissertation makes a contribution to Software Engineering because it is a well grounded empirical study that can assist those seeking to take software development seriously as a collaborative endeavor. Some of the most enduring studies in Software Engineering, such as Brook's law (Brooks, 1975), come out of a combination of participation and theorizing as displayed in this dissertation. In particular the theory advanced here may provide an analytically sound basis for understanding why agile software development is such a motivating alternative to the pre-planned specification of the waterfall model.

Beyond these contributions the archival method used in the replication study is a potentially useful contribution for those attempting to make sense of activity that is recorded in growing online archives. These archives provide evidence about sociality from projects as well known as Wikipedia to small online health support communities. Similarly, massive collaboration supported by internet technologies are touted as being important for political negotiation and democracy. The study of these phenomena can learn from the archival method in this dissertation, especially as it comes to separating archives-as-documentary-evidence and archives-as-explicit-actions.

Finally, as discussed in the Introduction, all of the data and intermediate artifacts for this dissertation are being made publicly available, through the FLOSSmole project (Howison et al., 2006). The Task catalogue may provide an advanced basis for further studies of FLOSS, as well as a teaching resource for those interested in showing how work actually unfolds in FLOSS projects. The ontologies and data representations developed in the course of the research also illustrate the promise of RDF and OWL for effective cross-repository data linking (Howison, 2008).

9.2 Future Research

The findings and theory of this dissertation suggest four courses of future research: Formalizing and testing the model, confirming empirical findings, detailed studies of FLOSS adaptation attempts and exploration of strategic deferral in other environments.

Formalizing and Testing the Model

The model advanced in this dissertation can be further formalized leading to greater clarity and certainty about its predictions. It should be possible to examine the combined effects of the individual decisions on the dynamics of a project through a multi-agent simulation. The simulation would allow testing the effect of different parameters, such as the motivational/confidence effects of success and failure in tasks. Feedbacks between multiple turns could reveal unexpected non-linear behaviors, especially once one is able to model use and recruitment effects. The results of work could be modeled in a way that would allow comparison to real empirical data on the success of FLOSS projects. Such a model would allow formal exploration of the "hare and tortoise" suggestions in Chapter 8 as well as quantifying the needed size of parameters for the predictions to match empirical findings.

The motivational effects of the experience of production are at the heart of this dissertation, yet they are not tested directly. It would be excellent to have empirical data for the hypothesis of differential effects from individual success and Co work failures. Since this study would have to proceed over time it should be designed as a panel study, possibly using a combination of diaries and surveys, such as those developed for testing the job design models discussed on page 146. Such a study would be invasive and require serious commitment from FLOSS developers, as such it might be best to conduct it with the blessing of a major umbrella organization like Debian or the Apache Foundation. It might also be possible to examine the effects in a laboratory study, using concocted teams and tasks.

One of the missing features of this model is that there is little insight into why Co work occurs at all. The model merely offers suggestions that it relates to the complexity and desirability of the task and might be supported by a history of observed successes of others which demonstrate skill and commitment, building cognitive and affective trust. The literature on the origins and stability of organizational routines (Gersick and Hackman, 1990; Feldman, 2000; Orlikowski and Yates, 1994; Becker, 2004) suggests other potential determinants, such as importation from other environments (recalling that FLOSS is embedded in other cultures of software development) and emphasizing path-dependency effects which might play an important role in a project's ability to balance co- and individual work during an exogenous ebb of developer availability and attention.

As discussed in the Discussion the question of the role of structural modularity in the codebase should be examined further, to see if the incremental individual work reported here relates to traditional metrics of coupling and cohesion.

Confirmation of empirical findings

The work would benefit from an improvement and expansion of the empirical results of the task-level findings reported in Chapter 5. The findings could be improved by the introduction of a reliability study, and expanded through study of other inter-release periods and, of course, more projects. It would be excellent to find content analytic coders able to replicate the archival reconstruction, and Chapter 5 provides a number of suggestions to that end. Unfortunately the ability to read code and to analyze behavior is a relatively rare combination. However when the work is publicized it might be the case that FLOSS developers themselves are motivated to improve the work and willing to spend time being trained, or other coders might be found in the relatively small number of researchers interested in empirical and organizational studies of software engineering.

Fire and Gaim were studied during successful phases of the projects. It would also be desirable to examine the reliability of the findings across the lives of these projects, by coding data from additional inter-release periods, especially as Fire begins to decline in effectiveness. Furthermore it would be interesting to examine the archives of critical cases of FLOSS development, such as Linux, to see if the method can provide insight into the task-level organization of these projects at different phases of their existence. Finally it would be worthwhile examining projects that undergo a change in some of the conditions identified as important for the operation of the model. For example one could examine a move from a community-based origin to a hybrid structure, such as when a commercial entity is formed and employs key participants. The converse would also be interesting, when a commercial entity spins out a project as FLOSS, such as the Firefox project. Such studies of hybrid situations would be strongly enriched by the understanding of more "pure" cases of community-based FLOSS presented in this dissertation.

Detailed studies of FLOSS adaptation

The model provides insight into why particular features of the FLOSS environment are crucial to FLOSS organization and the Discussion (Chapter 8) used this to examine three environments which have attempted to adapt FLOSS organizational techniques. These are currently little more than speculations and would benefit from extended empirical inquiries adopting the task operationalizations and analysis from Chapter 5. The archives of Wikipedia provide the potential for a task level analysis, as do the archives of both Open Hardware and Policy advocacy efforts. Care would need to be taken to ensure that the concept of Task was specified with enough contextual or emic understanding of the work, developed through interviews or through participation. The literature is sorely in need of structured comparative studies of these similar seeming instances of technology-supported distributed massive collaboration. Unfortunately most studies are, at best, similar to this dissertation in that they examine one phenomenon in detail and compare the other based on a limited empirical understanding.

Another possibility for examining the adaptation of FLOSS organizational techniques is to pursue action research with groups attempting to move their work towards a FLOSS model. Action research would design interventions, such as the introduction of particular technologies and practices designed to make work more additive and open to layered production. Such work may provide insight into the process of adaptation itself, whereby path effects in organizational change towards a FLOSS model could be further understood. Such path effects might be crucial to successful change.

Examination of "Strategic Deferral" in other environments

Finally it would be worthwhile taking up the idea of "strategic deferral" and examining whether it occurs in other contexts. For many years the production choices of traditional for-profit organizations have been framed as a choice between "build or buy", either developing an innovative product internally or purchasing an innovation through the market. Perhaps that choice could, in some circumstances, now be framed as "build, buy or wait"?

Deferral is sometimes successful in FLOSS because the landscape—the codebase—is constantly changing and provides new resources to understand and implement production differently. The effort and risk of the 'hard way' is avoided, with the trade-off being delay. Sometimes the situation never changes, sometimes changes may even make the innovation more difficult to conceptualize and implement. But sometimes unconnected, unplanned changes make work easier. It seems possible that this process could play out in other environments, where the benefit of being "first to market" is outweighed by the risky effort to build internally or the cost of purchasing. Instead, in fields with rapidly changing resource profiles, it might make strategic sense to conserve capital (or effort) until the business problem is easier to solve. Indeed the success of RedHat in building a business on top of and utilizing the ecosystem of open source (as well as contributing back to it) suggests that deferral and a kind of "strategic grazing" to package the results of massive voluntary collaboration might be the most flexible strategy for benefiting from FLOSS organizing in other domains.

9.3 In conclusion

The development and success of community-based FLOSS development is as interesting as it is surprising. This dissertation suggests that its success is the result of fit between the motivations of participants, the affordances of the informationalized task and the socio-technical infrastructure of work and organizing. A key overarching process making this form of organization possible is the informationalization of human activity together with a pervasive excess capacity in communication infrastructures. These processes do seem likely to continue and to pervade other areas of human activity. To the extent that they do there are good possibilities for adapting these ways of organizing to other kinds of work.

Yet the work in this dissertation sounds a strong note of caution in that assessment. The components of the socio-technical configuration described here reinforce each other, creating a bundle which works in its place, but which may prove tricky to 'unpick' for piecemeal adaptation in other kinds of work. Organizations seeking to harness the power of individual layered work may, for example, have to learn to tolerate or embrace practices of deferral.

The organization of community-based FLOSS development seems to provide a way forward for work outside the boundaries of traditional invest-and-control organization, offering autonomous, satisfying and useful work. Working together alone—individually in company—may offer a way forward, a way out of the traps of exploitation and alienation common across many fields of human endeavor. Yet it will not be a simple process, and much innovation—both technical and social—will be required to realize this potential. If this can be done—and it is far from easy—FLOSS may indeed realize its promise of a truly optimistic and emancipatory future of work.

Bibliography And Appendices

Details of Illustrative Tasks

Figure 1: Fire Task 2: Manual Browser Security Fix

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#gaim_period1_0002
Label: manual browser security fix
Start: Jul 20 2002 17:32:12 GMT+0000
End: Aug 25 2002 02:42:57 GMT+0000
Duration: 35D 9h 10m 45s
Actor Order: [ kareemy ] [ lschiere ] [ robot101 ] [ seanegan ] [ seanegan ] [ chipx86 ] [ seanegan ]
Action Order: [ useInformationProvision ] [ useInformationProvision ] [ coreProductionWork ] [ reviewWork ] [
polishingProductionWork ] [ coreProductionWork ] [ managementWork ]
Memo: Looks like seanegan applied a patch supplied by Robert McQueen, from a Bug report by Kareem Dana, commented on by Luke.

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_72
<kareemy> *reports bug with manual browser* (Jul 20 2002 17:32:12 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ TrackerSubmissionEvent ]
DocList: [ http://sourceforge.net/tracker/?group_id=235&atid=100235&func=detail&aid=584252#submission_document ]
ActionCodeApplied: [ task-codes:useInformationProvision ]
Details: Kareem Dua reports the bug that is eventually fixed in this Task
Memo:

  Gap:1D 5h 50m 1s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_43
<lschiere> *assists with bug diagnosis* (Jul 21 2002 23:22:13 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ TrackerCommentEvent ]
DocList: [ http://sourceforge.net/tracker/?group_id=235&atid=100235&func=detail&aid=584252#comment_375242_document ]
ActionCodeApplied: [ task-codes:useInformationProvision ]
Details: Luke Schiere replies to bug report with off the cuff assistance with diagnosis; he is actually wrong
Memo:

  Gap:20D 9h 41m 18s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_150
<robot101> *writes and conveys patch (implied)* (Aug 11 2002 09:03:31 GMT+0000)
Type: task:DatedImpliedAction
EventTypeList: [ SoftwareReleaseEvent ]
DocList: [ http://sourceforge.net/project/shownotes.php?release_id=107126group_id235#change_note ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: The change note thanks Robert McQueen for this fix, so this is an ImpliedAction as a place holder for McQueen's action.
Memo: ImpliedAction, note that this means the hasTime is approximate. This Action is implied from the thanks_quote of the Task
outcome, I can't find archival evidence for the Patch.  Linked by full RealName: Robert McQueen == robot101

  Gap:900 ms
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_176
<seanegan> *checks in manual browser fix* (Aug 11 2002 09:03:32 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3412#document ]
ActionCodeApplied: [ task-codes:reviewWork ]
Details: Sean Egan checks in the primary fix to the manual browser
Memo:

  Gap:10D 18h 10m 7s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_110
<seanegan> *tweaks manual browser fix* (Aug 22 2002 03:13:39 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3436#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:

  Gap:1D 20h 8m 47s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_34
<chipx86> *alters McQueen/Sean's fix* (Aug 23 2002 23:22:26 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ] [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3440#document ] [
http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3441#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: chip alters robot101/Sean's fix, making it much shorter and, presumably, more robust. This occurs on the Trunk
Memo: Merged two documents < 2 hours apart

  Gap:1D 3h 20m 31s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_108
<seanegan> *moves fix to branch for release* (Aug 25 2002 02:42:57 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3449#document ]
ActionCodeApplied: [ task-codes:managementWork ]
Details: Sean moves Chip's fix, which is a shorter replacement for the one he checked in, to the branch.
Memo: The SVN checkin here is Sean acknowledging that chip's fix is better than the one he checked in. Open question as to
whether this is part of the same Task, perhaps it's actually part of the Task to create the branch?
```

Figure 2: Gaim Task 34: TOC and ICQ Plugin removed, and Fire Task 57: user list duplicate fix

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#gaim_period1_0034
Label: TOC and ICQ plugin removed
Start: Aug 07 2002 23:25:33 GMT+0000
End: Aug 10 2002 02:42:55 GMT+0000
Duration: 2D 3h 17m 22s
Actor Order: [ seanegan ] [ chipx86 ] [ chipx86 ]
Action Order: [ coreProductionWork ] [ polishingProductionWork ] [ polishingProductionWork ]
Memo: Sean removes TOC and the ICQ plugin (both are old access methods, retired by OSCAR).  He does some
resulting cleanup of the build system and documents it in ChangeLog.  A little later Chip fixes some errors in
Sean's changes

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_64
<seanegan> *removes TOC and ICQ plugin from build process* (Aug 07 2002 23:25:33 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ] [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3401#document ] [
http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3402#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details:
Memo:

  Gap:18h 16m 35s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_36
<chipx86> *reverts a portion of Sean's changes, fixing the build* (Aug 08 2002 17:42:08 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3406#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:

  Gap:1D 9h 47s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_75
<chipx86> *fixes a bug in Sean's removal of TOC* (Aug 10 2002 02:42:55 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3409#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:
```

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#fire_period1_57
Label: user list duplicate fix
Start: Dec 06 2002 04:06:13 GMT+0000
End: Dec 06 2002 04:06:13 GMT+0000
Duration: --zero--
Actor Order: [ gbooker ]
Action Order: [ coreProductionWork ]
Memo: gb fixes a bug with user list duplicates.  Only the CVS checkin is found, not communications related to
it

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_372
<gbooker> *fixes user list duplicates bug* (Dec 06 2002 04:06:13 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1843#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details:
Memo:
```

Figure 3: Gaim Task 3: iconv library integrated and Gaim Task 7-5: change default options

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#gaim_period1_0003
Label: iconv library integrated
Start: Aug 02 2002 04:33:50 GMT+0000
End: Aug 02 2002 05:19:52 GMT+0000
Duration: 46m 2s
Actor Order: [ seanegan ] [ seanegan ] [ seanegan ]
Action Order: [ coreProductionWork ] [ documentationWork ] [ coreProductionWork ]
Memo: Thread http://sourceforge.net/tracker/?group_id=235&atid=300235&func=detail&aid=576704 might be relevant,
as might http://sourceforge.net/tracker/?group_id=235&atid=100235&func=detail&aid=486037.  Can't find the
relevant SVN commit; presumably it will be a left over, when I will search for the author of the release notes.
Maybe: http://pidgin.svn.sourceforge.net/viewvc/pidgin/trunk/gaim/src/core.h?r1=3386&r2=3385&pathrev=3386

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_19
<seanegan> *checks in iconv* (Aug 02 2002 04:33:50 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3380#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: Sean checks in code that draws on iconv for international character conversion; the SVN commit message
'We've returned' perhaps indicates a return from post-release slow down?
Memo:

  Gap:19m 52s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_44
<seanegan> *announces iconv via ChangeLog* (Aug 02 2002 04:53:42 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3387#document ]
ActionCodeApplied: [ task-codes:documentationWork ]
Details: Sean announces the intergration of iconv via the ChangeLog
Memo: The SVN commit message indicates that the ChangeLog is the communication system for code changes

  Gap:26m 10s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_117
<seanegan> *intergrates iconv library* (Aug 02 2002 05:19:52 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ] [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3389#document ] [
http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3390#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: Sean Egan checks in more code changes that integrate iconv; the second check-in is a bugfix for the
first.
Memo: Three CVS commits put together as an action because they are < 2 hours apart


Ep: http://floss.syr.edu/ontologies/2008/task-data#gaim_period1_0007_5
Label: change default options
Start: Aug 02 2002 04:52:48 GMT+0000
End: Aug 02 2002 04:53:42 GMT+0000
Duration: 54s
Actor Order: [ seanegan ] [ seanegan ]
Action Order: [ coreProductionWork ] [ documentationWork ]
Memo: seanegan Changes some default options The change note change doesn't in make it to 0.59.1

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_23
<seanegan> *checks in change to default options* (Aug 02 2002 04:52:48 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3386#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: Sean, in amongst a set of other changes, checks in a change of default options
Memo:

  Gap:54s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_82
<seanegan> *changes ChangeLog for default options* (Aug 02 2002 04:53:42 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3387#document ]
ActionCodeApplied: [ task-codes:documentationWork ]
Details: In amongst a set of changes (mostly from 3386), Sean includes the change of default options
Memo:
```

Figure 4: Fire Task 29: aim buddy icons

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#fire_period1_29
Label: aim buddy icons
Start: Oct 27 2002 02:58:32 GMT+0000
End: Nov 04 2002 01:44:40 GMT+0000
Duration: 7D 22h 46m 8s
Actor Order: [ gbooker ] [ gbooker ] [ gbooker ] [ gbooker ] [ gbooker ] [ gbooker ] [ gbooker ] [ gbooker ]
Action Order: [ coreProductionWork ] [ documentationWork ] [ polishingProductionWork ] [ polishingProductionWork ] [
polishingProductionWork ] [ polishingProductionWork ] [ polishingProductionWork ] [ polishingProductionWork ]
Memo: gbooker adds buddy icons; this seems to be related to libfaim, but also related to the file transfer code in someway?

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_216
<gbooker> *documents completion of buddy icons* (Oct 27 2002 02:58:32 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1574#document ]
ActionCodeApplied: [ task-codes:documentationWork ]
Details:
Memo:
  Gap:--zero--
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_503
<gbooker> *checks in buddy icon receiving code* (Oct 27 2002 02:58:32 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1574#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details:
Memo:
  Gap:39m 3s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_59
<gbooker> *polishes buddy icons, adding jpg* (Oct 27 2002 03:37:35 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1575#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:
  Gap:1h 22m 22s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_413
<gbooker> *polishes buddy icons, adding bitmaps* (Oct 27 2002 04:59:57 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1576#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:
  Gap:12h 1m 39s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_12
<gbooker> *polishing buddyicons* (Oct 27 2002 17:01:36 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1577#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details: save as .buddyicon
Memo:

  Gap:3D 13h 1m 50s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_550
<gbooker> *fixes buddy icon for IRC* (Oct 31 2002 06:03:26 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1594#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:
  Gap:3D 18h 34m 51s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_526
<gbooker> *polishing buddy icons* (Nov 04 2002 00:38:17 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1641#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details: memory related
Memo:
  Gap:1h 6m 23s
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Fire1DocToEpMapping_144
<gbooker> *polishing buddy icons (memory fix)* (Nov 04 2002 01:44:40 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://fire.svn.sourceforge.net/viewvc/fire?view=rev&revision=1644#document ]
ActionCodeApplied: [ task-codes:polishingProductionWork ]
Details:
Memo:
```

Figure 5: Gaim Task 18: Finnish Translation

```
Ep: http://floss.syr.edu/ontologies/2008/task-data#gaim_period1_0018
Label: Finnish Translation
Start: Jul 02 2002 23:51:04 GMT+0000
End: Jul 02 2002 23:51:05 GMT+0000
Duration: 1 ms
Actor Order: [ teroajk ] [ robflynn ] [ robflynn ]
Action Order: [ coreProductionWork ] [ reviewWork ] [ assigingCredit ]
Memo: robflynn adds the Finnish Translation, to the 0.60 changenote area.

Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_164
<teroajk> *updates Finnish Translation* (Jul 02 2002 23:51:04 GMT+0000)
Type: task:UnDatedImpliedAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3364#document ]
ActionCodeApplied: [ task-codes:coreProductionWork ]
Details: Rob Flynn thanks Tero Kuusela in the ChangeLog; therefore he must have updated the Translation.
Memo:

  Gap:1 ms
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_14
<robflynn> *thanks Finnish translator for work.* (Jul 02 2002 23:51:05 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3364#document ]
ActionCodeApplied: [ task-codes:assigingCredit ]
Details:
Memo:

  Gap:--zero--
Action: http://floss.syr.edu/ontologies/2008/task-data#action_Gaim1DocumentToEpisodeMapping_113
<robflynn> *integrates Finnish translation* (Jul 02 2002 23:51:05 GMT+0000)
Type: task:ExplicitAction
EventTypeList: [ SvnCommitEvent ]
DocList: [ http://gaim.svn.sourceforge.net/viewvc/gaim?view=rev&revision=3364#document ]
ActionCodeApplied: [ task-codes:reviewWork ]
Details: Rob Flynn checks updated Finnish Translations into SVN
Memo:
```

Bibliography

Ambrose, M. L. and Kulik, C. T. (1999). Old friends, new faces: Motivation research in the 1990s. *Journal of Management*, 25:231–292.

Annabi, H. (2005). *Moving from individual contribution to group learning the early years of the Apache Web server.* PhD thesis, Syracuse University.

Annabi, H., Crowston, K., and Heckman, R. (2008). Depicting what really matters: Using episodes to study latent phenomenon. In *Proceedings of the International Conference on Information Systems (ICIS)*, Paris.

Arrow, H., McGrath, Joseph E., J. E., and Berdahl., J. L. (2000). *Small Groups as Complex, Systems: Formation, Coordination, Development, and Adaptation.* SAGE Publishing Co, Thousand Oaks, CA.

Bachrach, D. G., Powell, B. C., Collins, B. J., and Richey, R. G. (2006). Effects of task interdependence on the relationship between helping behavior and group performance. *Journal of Applied Psychology*, 91(6):1396–1405.

Baldwin, C. Y. and Clark, K. (2001). *Design Rules: The Power of Modularity.* Harvard Business School Press, Cambridge, MA.

Bandura, A. (1997). *Self-efficacy: The exercise of control.* Freeman, New York.

Barley, S. R. (1986). Technology as an occasion for structuring: observations on ct scanners and the social order of radiology depart-ments. *Administrative Science Quarterly*, 31:78–108.

Barnard, C. I., Andrews, K. R., and Andrews, K. R. (1968). *The Functions of the Executive.*

Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherl, J., and Thomas, D. (2001). Manifesto for agile software development. Online manifesto, Non-Profit Agile Alliance.

Becker, M. C. (2004). Organizational routines: A review of the literature. *Industrial and Corporate Change*, 13(4):643–678.

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):183–194.

Bertolotti, F., Macrì, D., and Tagliaventi, M. (2005). Spontaneous self-managing practices in groups: evidence from the field. *Journal of Management Inquiry*, 14:366–385.

Bezroukov, N. (1999). A second look at the cathedral and bazaar. *First Monday*, 4(12).

Bowen, G. A. (2006). Grounded theory and sensitizing concepts. *International Journal of Qualitative Methods*, 5(3):1–9.

Brooks, F. P. (1975). *The Mythical Man-Month: Essays on Software Engineering*, chapter The Mythical Man Month, pages 13–29. Addison-Wesley Pub Co.

Burns, M. and Howison, J. (2001). Napster fabbing: Internet delivery of physical products. *Rapid Prototyping Journal*, 7(4):194–196.

Chin, P. O. and Cooke, D. (2004). Satisfaction and coordination in virtual communities. In *Proceedings of the Tenth Americas Conference on Information Systems*, New York, New York.

Cogburn, D. L., Bhattacharyya, S., Sharif, R. M., Johnsen, J. F., and Howison, J. (2005). Distributed deliberative citizens: Exploring the impact of policy collaboratories on transnational ngo network participation in wsis. In *Proc. of ICA Conference of the Americas*.

Conklin, M. (2004). Do the rich get richer? The impact of power laws on open source development projects. In *Proceedings of Open Source 2004 (OSCON)*, Portland, Oregon.

Conley, C. A. (2008). *Design for quality: The case of Open Source Software Development*. Unpublished doctoral dissertation, New York University, New York, NY.

Conway, M. E. (1968). How do committees invent? *Datamation*, 149(4329):28–31.

Cook, S. (2008). The contribution revolution: Letting volunteers build your business. *Harvard Business Review*.

Crowston, K. (2003). A taxonomy of organizational dependencies and coordination mechanisms. In Malone, T. W., Crowston, K., and Herman, G., editors, *Tools for Organizing Business Knowledge: The MIT Process Handbook*. MIT Press, Cambridge, MA.

Crowston, K., Annabi, H., and Howison, J. (2003). Defining open source software project success. In *ICIS 2003. Proceedings of International Conference on Information Systems 2003*.

Crowston, K. and Fagnot, I. (2008). The motivational arc of massive virtual collaboration. In *Proceedings of the IFIP WG 9.5 Working Conference on Virtuality and Society: Massive Virtual Communities*, Lüneberg, Germany.

Crowston, K. and Howison, J. (2006). Assessing the health of open source communities. *IEEE Computer*, 39(5):89–91.

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

Crowston, K. and Osborn, C. S. (2003). A coordination theory approach to process description and redesign. In Malone, T. W., Crowston, K., and Herman, G. A., editors, *Organizing Business Knowledge*, pages 335–369. The MIT Press, Cambridge, MA.

Crowston, K., Rubleske, J., and Howison, J. (2006b). Coordination theory and its application in HCI. In Zhang, P. and Galletta, D., editors, *Human-Computer Interaction in Management Information Systems: Foundations*, volume 5 of *Advances in Management Information Systems*, pages 120–140. M. E. Sharpe, Inc, Armonk, NY.

Crowston, K. and Scozzi, B. (2004). Coordination practices for bug fixing within FLOSS development teams. In *First International Workshop on Computer Supported Activity Coordination (CSAC 2004)*, Porto (Portugal).

Crowston, K., Wei, K., Howison, J., and Wiggins, A. (2008). Free/libre open source software development: What we know and what we do not know. Under Review.

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.

Drucker, P. (2002). They're not employees, they're people. *Harvard Business Review*, 80(2):70–77.

Faraj, S. and Xiao, Y. (2006). Coordination in fast-response organizations. *Management Science*, 52(8):1155–1169.

Feldman, M. S. (2000). Organizational routines as a source of continuous change. *Organization Science*, 11(6):611–629.

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

Fielding, R. T. (1999). Shared leadership in the Apache project. *Association for Computing Machinery. Communications of the ACM*, 42(4):42.

Fogel, K. (1999). *Open Source Development with CVS.* Coriolis Open Press, Scottsdale, AZ.

Gacek, C. and Arief, B. (2004). The many meanings of open source. *IEEE Software*, 21(1):34–40.

Gao, Y., Antwerp, M. V., Christley, S., and Madey, G. (2007). A research collaboratory for open source software research. In *the Proceedings of the 29th International Conference on Software Enginering + Workshops (ICSE-ICSE Workshops 2007).* Minneapolis, MN.

Gersick, C. J. G. and Hackman, J. R. (1990). Habitual routines in task-performing groups. *Organizational Behavior and Human Decision Processes*, 47:65–97.

Ghosh, R. A., Robles, G., and Glott, R. (2002). Free/libre and open source software: Survey and study floss. Technical report, International Institute of Infonomics, University of Maastricht: Netherlands.

Glaser, B. G. (1978). *Theoretical sensitivity: Advances in the methodology of grounded theory.* Sociology Press, Mill Valley, CA.

Glaser, B. G. and Strauss, A. L. (1967). *The discovery of grounded theory: strategies for qualitative research.* Aldine, Chicago.

González-Barahona, J. M. and Robles-Martínez, G. (2003). Unmounting the "code gods" assumption. In *Proceedings of the Fourth International Conference on eXtreme Programming and Agile Processes in Software Development (XP2003).*

Greg Madey (ed) (2007+). The sourceforge research data archive (SRDA). http://zerlot.cse.nd.edu/.

Haas, E. B., Williams, M. P., and Babai, D. (1977). *Scientists and world order: The uses of technical knowledge in international organizations.* University of California Press, Berkeley, CA.

Hackman, J. R. and Morris, C. G. (1978). Group tasks, group interaction process, and group performance effectiveness: A review and proposed integration. In Berkowitz, L., editor, *Group Processes*, volume 8 of *Advances in Experimental Social Psychology*, pages 45–99. Academic Press, New York.

Hackman, J. R. and Oldham, G. R. (1980). *Work Redesign.* Addison- Wesley, Reading, Mass.

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

Harrison, W. (2001). Editorial: Open source and empirical software engineering. *Empirical Software Engineering*, 6(3):193–194.

Hars, A. and Ou, S. (2002). Working for free? Motivations of participating in FOSS projects. *International Journal of Electronic Commerce*, 6(3):25–39.

Heckman, R., Crowston, K., Li, Q., Allen, E., Eseryel, U. Y., Howison, J., and Wei, K. (2006). Emergent decision-making practices in technology-supported self-organizing distributed teams. In *ICIS 2006. Proceedings of International Conference on Information Systems 2006*.

Hemetsberger, A. (2001). Fostering cooperation on the internet: social exchange processes in innovative virtual consumer communities. In *Proceedings of the Association for Consumer Research (ACR) 2001*, Austin, Texas. 02.

Herbsleb, J. D. and Grinter, R. E. (1999). Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70.

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

Hertel, G., Niedner, S., and Herrmann, S. (2003). Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research Policy*, 32(7):1159–1177.

Herzberg, F. (1966). *Work and the nature of man.* World Publishing, Cleveland.

Hill, M. R. (1993). *Archival Strategies and Techniques.* Number 31 in Qualitative Research Methods. Sage Publications.

Howe, J. (2006). The rise of crowd-sourcing. *Wired Magazine*, 14(6).

Howison, J. (2008). Cross-repository data linking with rdf and owl. towards common ontologies for representing floss data. In *Proc. of WoPDaSD (Workshop on Public Data at International Conference on Open Source Software)*.

Howison, J., Conklin, M., and Crowston, K. (2006). FLOSSmole: A collaborative repository for FLOSS research data and analysis. *International Journal of Information Technology and Web Engineering*, 1(3):17–26.

Howison, J. and Goodrum, A. (2004). Why can't I manage academic papers like MP3s? The evolution and intent of metadata standards. In *Proc. of the 2004 Colleges, Code and Intellectual Property Conference*, number 57 in ACRL Publications in Librarianship, College Park, MD.

Howison, J., Wiggins, A., and Crowston, K. (2008). eResearch workflows for studying free and open source software development. In *Proceedings of the Fourth International Conference on Open Source Software (IFIP 2.13)*.

Hughes, J., Lang, K. R., Clemons, E. K., and Kauffman, R. J. (2008). A unified interdisciplinary theory of open source culture and entertainment. Technical Report 1077909, SSRN.

Humphrey, S. E., Nahrgang, J. D., and Morgeson, F. P. (2007). Integrating motivational, social, and contextual work design features: A meta-analytic summary and theoretical extension of the work design literature. *Journal of Applied Psychology*, 92(5):1332–1356.

Ilgen, D., Hollenbeck, J., Johnson, M., and Jundt, D. (2005). Teams in organizations: From input-process-output models to IMOI models. *Annual Review of Psychology*, 56(1):517–543.

Jensen, C. and Scacchi, W. (2007). Guiding the discovery of open source software processes with a reference model. In Feller, J., Fitzgerald, B., Scacchi, W., and Sillitti, A., editors, *Open Source Development, Adoption and Innovation*, volume 234 of *IFIP International Federation for Information Processing*, pages 265–270. Springer, Boston, USA.

Ke, W. and Zhang, P. (2008). Participating in open source software projects: The role of empowerment. In *Proceedings of ICIS08 HCI Workshop*.

Kellogg, K., Orlikowski, W., and Yates, J. (2006). Life in the trading zone: Structuring coordination across boundaries in postbureaucratic organizations. *Organization Science*, 17(1):22–44.

Kiggundu, M. N. (1981). Task interdependence and the theory of job design. *Academy of Management Review*, 6:499–508.

Kiggundu, M. N. (1983). Task interdependence and job design: Test of a theory. *Organizational Behavior and Human Performance*, 31:145–172.

Kittur, A., Chi, E., Pendleton, B. A., Suh, B., and Mytkowicz, T. (2007). Power of the few vs. wisdom of the crowd: Wikipedia and the rise of the bourgeoisie. In *Alt.CHI, 2007*.

Kittur, A. and Kraut, R. E. (2008). Harnessing the wisdom of crowds in wikipedia: Quality through coordination. In *CSCW 2008: Proceedings of the ACM Conference on Computer-Supported Cooperative Work*, New York. ACM Press.

Kozlowski, S. W. J., Hully, S. M., Nason, E. R., and Smith, E. M. (1999). Developing adaptive teams: A theory of compilation and performance across levels and time. In IIgen, D. R. and Pulakos, E. D., editors, *The changing nature of performance*, pages 240–292. Jossey-Bass Publishers, San Francisco.

Krikelas, J. (1983). Information seeking behaviour: Patterns and concepts. *Drexel Library Quarterly*, 19(2):5–20.

Krishnamurthy, S. (2002). Cave or community: An empirical examination of 100 mature open source projects. *First Monday*, 7(6).

Kuznetsov, S. (2006). Motivations of contributors to Wikipedia. *ACM SIGCAS Computers and Society*, 36(2).

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

Lakhani, K. and Wolf, R. (2003). Why hackers do what they do: Understanding motivation efforts in Free/F/OSS projects. Working Paper 4425-03, MIT Sloan School of Management.

Langlois, R. N. (2002). Modularity in technology and organization. *Journal of Economic Behavior & Organization*, 49(1):19–37.

Lessig, L. (2002). *The Future of Ideas: The fate of the commons in a connected world.* Random House.

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

Locke, E. A. (1996). Motivation through conscious goal setting. *Applied and Preventive Psychology*, 5:117–124.

Locke, E. A. and Latham, G. P. (1990). *A theory of goal setting and task performance.* Prentice-Hall, Englewood Cliffs, NJ.

Luthiger, B. (2004). Fun and software development (FASD) study provisional results. Progress report, Universitat Zurich.

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

Malone, T. W. (2004). *The Future of Work: How the New Order of Business Will Shape Your Organization, Your Management Style and Your Life.* MIT Press.

March, J. G. and Simon, H. A. (1958). *Organizations.* Blackwell, Oxford, UK.

Marks, M. A., Mathieu, J. E., and Zaccaro, S. J. (2001). A temporally based framework and taxonomy of team processes. *Academy of Management Review*, 26(3):356–377.

Marvin, C. (1988). *When Old Technologies Were New.* Oxford University Press, New York.

McGrath, J. (1984). *Groups: Interaction and Performance.* Prentice-Hall, Englewood Cliffs, NJ.

McGrath, J. E. (1991). Time, interaction, and performance (TIP): A theory of groups. *Small Group Research*, 22:147–174.

McGrath, J. E. and Tschan, F. (2004). Dynamics in groups and teams: Groups as complex action systems. In Poole, M. S. and Van de Ven, A. H., editors, *Handbook of Organizational Change and Innovation.* Oxford University Press, Oxford, UK.

Michlmayr, M. (2003). Quality and the reliance on individuals in free software projects. In *Proceedings of the ICSE 3rd Workshop on Open Source*.

Michlmayr, M. (2004). Managing volunteer activity in free software projects. In *Proceedings of the 2004 USENIX Annual Technical Conference, FREENIX Track*, pages 93–102, Boston, USA.

Michlmayr, M. (2005). Quality improvement in volunteer open source projects. Research plan (shared for IntOSS consortium).

Miles, M. B. and Huberman, A. (1984). *Qualitative Data Analysis: A Sourcebook of New Methods*. Sage, Beverly Hills, CA.

Mintzberg, H. (1979). *The Structuring of Organizations*. Prentice-Hall, Englewood Cliffs, NJ.

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):309–346.

Nakakoji, K. and Yamamoto, Y. (2001). Taxonomy of open source software development. In *Proceedings of the ICSE 1st Workshop on Open Source*.

Nov, O. (2007). What motivates Wikipedians?

Olson, J. S., Teasley, S., Covi, L., and Olson, G. M. (2002). The (currently) unique advantages of collocated work. In Hinds, P. and Kiesler, S., editors, *Distributed Work*, pages 113–135. MIT Press, Cambridge, MA.

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 Barley, S. R. (2002). Technology and institutions: What can research on information technology and research on organizations learn from each other? *MIS Quarterly*, 25(2):145–165.

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):121–145.

Orlikowski, W. J. and Yates, J. (1994). Genre repertoire—the structuring of communicative practices in organizations. *Administrative Science Quarterly*, 39(4):541–574.

Ortega, F., Gonzalez-Barahona, J. M., and Robles, G. (2008). On the inequality of contributions to wikipedia. In *Proceedings of the Proceedings of the 41st Annual Hawaii international Conference on System Sciences*.

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

Patton, M. Q. (2002). *Qualitative research and evaluation methods*. Sage, Thousand Oaks, CA, 3rd edition.

Perlow, L., Gittell, J., and Katz, N. (2004). Contextualizing patterns of work group interactions: Toward a nested theory of structuration. *Organization Science*, 15(5):520–536.

Poole, M. and Roth, J. (1989). Decision development in small groups iv: A typology of group decision paths. *Human Communication Research*, 15(3):323–356.

Porter, L. W. and Lawler, E. E. (1968). *Managerial attitudes and performance*. Irwin, Homewood, IL.

Powell, A., Piccoli, G., and Ives, B. (2004). Virtual teams: A review of current literature and directions for future research. *Database for Advances in Information Systems*, 35(1):6.

Raymond, E. S. (1998). The Cathedral and the Bazaar. *First Monday*, 3(3).

Raymond, E. S. and Moen, R. (2001). How to ask questions the smart way. Webpage.

Rico, R. and Cohen, S. G. (2006). Effects of task interdependence and communication technologies synchrony on performance in virtual teams. *Journal of Managerial Psychology*, 20(3/4):261–274.

Robbins, J. E. (2002). Adopting OSS methods by adopting OSS tools. In *Proceedings of the ICSE 2nd Workshop on Open Source*.

Robles, G. (2007). Research friendly: Community meets research. In *International Workshop on Public Data about Software Development*.

Rosenberg, S. (2007). *Dreaming in Code: Two Dozen Programmers, Three Years, 4,732 Bugs, and One Quest for Transcendent Software*. Crown.

Rousseau, V., Aube, C., and Savoie, A. (2006). Teamwork Behaviors: A Review and an Integration of Frameworks. *Small Group Research*, 37(5):540.

Ryan, R. M. and Deci, E. L. (2000). Intrinsic and extrinsic motivations: Classic definitions and new directions. *Contemporary Educational Psychology*, 25:54–67.

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

Scacchi, W. (2004). Free/open source software development practices in the computer game community. *IEEE Software*, 21(1):56–66.

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):351–365.

Scheier, I. (1977). Staff nonsupport of volunteers: A new look at an old failure. *Voluntary Action Leadership*, Fall.

Schoonhoven, C. B. (1981). Problems with contingency theory: Testing assumptions hidden within the language of contingency 'theory'. *Administrative Science Quarterly*, 26(3):349–377.

Schroer, J. and Hertel, G. (2006). Wikipedia: Motivation for voluntary engagement in an open web-based encyclopedia. In *8th International General Online Research Conference (March 2006), Bielefeld, Germany.*

Schweik, C. M. and English, R. (2007). Tragedy of the foss commons? investigating the institutional designs of free/libre and open source software projects. *First Monday*, 12(2).

Scialdone, M., Li, N., Howison, J., Heckman, R., and Crowston, K. (2008). Group maintenance in technology-supported distributed teams. In *Best Paper Proceedings of Academy of Management Annual Meeting*, Anaheim, CA.

Senyard, A. and Michlmayr, M. (2004). How to have a successful free software project. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference*, pages 84–91, Busan, Korea. IEEE Computer Society.

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

Simon, H. (1957). A behavioral model of rational choice. In *Models of Man, Social and Rational: Mathematical Essays on Rational Human Behavior in a Social Setting.* New York: Wiley.

Simon, H. A. (1960). *The New Science of Management Decision Making.* Evanston, New York.

Steers, R. M., Mowday, R. T., and Shapiro, D. L. (2004). The future of work motivation theory. *Academy of Management Review*, 29(3):379–387.

Stewart, K. J., Ammeter, A. P., and Maruping, L. M. (2006). Impacts of license choice and organizational sponsorship on user interest and development activity in open source software projects. *Information Systems Research*, 17(2):126–144.

Surowiecki, J. (2005). *The Wisdom of Crowds.* Anchor.

Taggar, S. and Haines, III, V. Y. (2006). I need you, you need me: A model of initiated task interdependence. *Journal of Managerial Psychology*, 21(3):211–230.

Tapscott, D. and Williams, A. D. (2006). *Wikinomics: How Mass Collaboration Changes Everything*. Portfolio Hardcover.

Thomas, E. (1957). Effects of facilitative role interdependence on group functioning. *Human Relations*, 10:347–366.

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

Trist, E. L. and Bamforth, K. W. (1951). Social and psychological consequences of the longwall method of coal-getting. *Human Relations*, 4(1):3–28.

Tschan, F. and von Cranach, M. (1996). Group task structure, processes, and outcome. In West, M. A., editor, *Handbook of work group psychology*, pages 92–121. John Wiley, Chichester, England.

Van de Ven, A. H., Delbecq, A. L., and Koenig, R. (1976). Determinants of coordination modes within organizations. *American Sociological Review*, 41(2):332–338.

Van Der Vegt, G., Emans, B., and Van De Vliert, E. (2000). Team members' affective responses to patterns of intragroup interdependence and job complexity. *Journal of Management*, 26:633– 655.

von Hippel, E. (1990). Task partitioning: an innovation process variable. *Research Policy*, 19(5):407–418.

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):209–223.

von Krogh, G., Spaeth, S., and Lakhani, K. R. (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241.

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

Wageman, R. (1995). Interdependence and group effectiveness. *Administrative Science Quarterly*, 40(1):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):687–700.

Warstaa, J. and Abrahamsson, P. (2003). Is open source software development essentially an agile method? In *Proceedings of the ICSE 3rd Workshop on Open Source*.

Williamson, O. E. (1981). The economics of organization: The transaction cost approach. *The American Journal of Sociology*, 87(3):548–577.

Winograd, T. and Flores, F. (1987). *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley Professional.

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)*.

Ye, Y. and Kishida, K. (2003). Toward an understanding of the motivation of open source software developers. In *Proceedings of 2003 International Conference on Software Engineering (ICSE)*, Portland, Oregon, USA.

Yin, R. (1994). *Case study research: Design and methods (2nd ed.)*. Sage Publishing, Beverly Hills, CA.

Zaheer, S., Albert, S., and Zaheer, A. (1999). Time scales and organizational theory. *Academy of Management Review*, 24(4):725–741.

Biographical Data

James Howison defended his Ph.D. at the School of Information Studies at Syracuse University in December 2008 and began a post-doctoral fellowship in the School of Computer Science at Carnegie Mellon University in January 2009. His research focuses on the organization of distributed collaboration, especially in Free and Open Source software projects. His publications include articles in IEEE Computer, IEEE Transactions on Professional Communications, Software Process Improvement and Practice as well as Knowledge, Technology and Policy. He has presented at the International Conferences for Information Systems (ICIS) and Software Engineering (ICSE). He was selected as a participant at the ICIS doctoral consortium in 2007 and the first NSF workshop on the Science of Socio-technical Systems in 2008. He has been invited to speak at O'Reilly's eTech, OSCON and FOOcamp conferences. Updated details can be found at http://james.howison.name.

Born in Scotland, James grew up in Australia, earning his undergraduate Economics degree from the University of Sydney and pursued masters study in Software Engineering at the University of New South Wales before transferring to the Syracuse Ph.D. in Information Science and Technology in 2002. Prior to returning to graduate school James worked as an information systems implementation consultant with KPMG management consultants and as a consultant with Control Risks Group, an international crisis management and corporate investigations consultancy. In 2005 James was chief scientist for a prize winning business plan competition team, organized by the top-ranked Entrepreneurship program at Syracuse.

James is a keen golfer and sailor, competing in national and international championships in the Etchells and Sydney 38 classes as well as over 1500 n.m. offshore experience. He is a keen supporter of Rugby and Cricket but is always open to American attempts to convince him of the value of North American sports.