12-2009

TOWARDS A THEORY ON THE SUSTAINABILITY AND PERFORMANCE OF FLOSS COMMUNITIES

Mohammad Almarzouq
Clemson University, malmarz@clemson.edu

TOWARDS A THEORY ON THE SUSTAINABILITY AND PERFORMANCE OF FLOSS COMMUNITIES

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Management

---

by
Mohammad AlMarzouq
December 2009

Accepted by:
Dr. Varun Grover, Committee Co-Chair
Dr. Jason Bennett Thatcher, Committee Co-Chair
Dr. DeWayne Moore
Dr. Richard Klein
Dr. Molly Wasko

Abstract

With the emergence of Free/Libre and Open Source Software as a significant force that is reshaping the software industry, it becomes more important to reassess conventionally held wisdom about software development. Recent literature on the FLOSS development process suggests that our previously held knowledge about software development might be obsolete. We specifically highlight the tension between the views embodied by the Linus' Law and Brooks' Law.

Linus' Law was forwarded by Eric Raymond and suggests that the FLOSS development process benefits greatly from large numbers of developers. Brooks' Law, which is part of currently held wisdom on software development, suggests that adding developers is detrimental to the progress of software projects. Raymond explains that the distributed nature of the FLOSS development process and the capacity of source code to convey rich information between developers are the main causes of the obsolescence Brooks' Law in the FLOSS development context.

By performing two separate studies, we show how both views of software development can be complementary. Using the lens of Transaction Cost Theory (TCT) in the first study, we identify the characteristics of the development knowledge as being the main factors constraining new members from contributing source code to FLOSS development projects. We also conceptualize of these knowledge characteristics as being analogous to what Brooks' described as the ramp-up effect. We forward the argument, and offer em-

pirical validation, that managing these characteristics of knowledge would result in an increase the number of contributors to a FLOSS projects.

The second study is concerned with the impact if having these new members added to the development team in a FLOSS project. Using the lens of Organizational Information Processing Theory (OIPT), we forward the argument, and offer empirical validation, that more contributors can be detrimental to progress if the committers of a FLOSS project are overwhelmed. Our findings also suggest that large development teams are indeed possible in FLOSS, however, they must be supported by proper source code design and community structures.

Dedication

To my parents and grand parents, who raised me to appreciate knowledge and set an example for me to follow on dedication and integrity. To my wife, Ruqayyah, and sons, Fahad and Abdulrahman, who endured with me and supported me throughout my years as a PhD student. I love you all.

Acknowledgement

Special thanks to Kuwait University and the Government of Kuwait for the scholarship that supported me throughout the PhD program. Special thanks also go to the attendees and organizers of the AOM 2006 FLOSS Workshop and the OSS 2009 PhD Consortium for their comments which helped me tremendously in shaping my thoughts into this work.

Table Of Contents

Table Of Figures

Table Of Tables

Chapter 1

Introduction

Free/Libre and Open Source Software (FLOSS) refers to software that has its source code made available for users to examine, use, and modify (Raymond, 2001; AlMarzouq et al., 2005). As a result of making the source code accessible, users may participate in the development of the product by uncovering and fixing bugs and security holes or implementing new features (Raymond, 2001). User participation in development is thought to increase the quality of the software; as Eric Raymond put it, "Given enough eyeballs, all bugs are shallow" (Raymond, 2001, p.30). In addition, having more users would put the software in different operating conditions that could expose hidden bugs in the implementation, which can then be fixed by any user willing to put forth the effort to do so (Raymond, 2001; Liu and Iyer, 2007).

While the improved quality is the most tangible benefit of the FLOSS development model, there seems to be a broader impact of this model, which is reshaping the dynamics of competition in the software industry (Economides and Katsamakas, 2006; Bonaccorsi et al., 2006). Take for example the recent move by Sun to release Java as FLOSS and leverage the FLOSS development model[1], or Microsoft's foray into FLOSS development with the establishment of the codeplex.com website for incubating FLOSS projects and

[1] See http://www.openjdk.org

providing tools for users and developers to collaborate[2]. In addition, the 2007 development report by the Linux Foundation suggests that over 11% of the changes made to the Linux Kernel source code were sponsored by some of the biggest companies in the software and technology industries, such as Intel, IBM, HP, Red Hat, Cisco, and Google (Kroah-Hartman et al., 2008).

These moves by for-profit organizations seem at odds with the traditional way these entities operate in the software industry. On one hand, we see bitter competitors cooperating on the development of the Linux Kernel and other FLOSS projects. On the other hand, we see organizations that have built their business on selling software licenses, releasing the software as FLOSS and relinquishing all of the profit that could potentially come from selling copies of the software (AlMarzouq et al., 2005). These two observations suggest that there is a fundamental shift happening in the software and technology industries, which we observe to be highly related to FLOSS.

The relationship between FLOSS and the shift in the software industry was highlighted by Tim O'Reilly's description of FLOSS as a "commoditizing" force. According to O'Reilly, what FLOSS is doing to the software industry is exactly what the PC did to the super computer industry (Schwartz et al., 2009). As a result of the FLOSS commoditizing force, established business models for selling software licenses are no longer as profitable, and new profitable business models are emerging that rely on commoditized software.

---

[2] See http://www.codeplex.com

For example, Riehle (2007) observed that software licensing was a significant cost component to companies offering turn-key solutions, such as IBM. FLOSS would allow such companies to eliminate the cost associated with software licensing and take in a bigger profit, while reducing the cost to the customer at the same time. Furthermore, one could attribute the recent emergence of network and "cloud" services to this commoditizing force of FLOSS (Schwartz et al., 2009). These services deliver computing resources to customers, which are built on commoditized hardware and software layers, through the Internet. By shielding the customer from the complexities of managing a computing infrastructure, the companies that offer cloud and network services create value for the customer and earn profit for themselves. In addition, these companies create value from the ease with which such services can be scaled up or down based on customer needs. Without the commoditized hardware and software layers, such infrastructures would have been expensive to create (Schwartz et al., 2009). As a result, profitability shifted from established business models of licensing infrastructure management software to service models that offered complete infrastructure solutions.

Thus far, we have described what we believe is a fundamental shift in the software industry resulting from the FLOSS movement. This highlights the importance of understanding FLOSS for both academics and practitioners alike. What we have not explained yet is how FLOSS brought about that change. To understand this how this change came about, we first need to give a brief introduction to the concepts and history surrounding FLOSS.

1 FLOSS: A Primer[3]

The origins of FLOSS date back to the computing industry in the 1950s. Back then, all software was free and all source code was accessible since the value of software had yet to be recognized. During the 1960s, the Department of Defense built ARPAnet, which connected researchers and engineers and lead to the establishment of the initial informal guidelines for distributed software development (Raymond, 2000a), which can be viewed as the early incarnation of FLOSS. The situation with software changed when IBM unbundled software from hardware, which lead to the recognition software's value. This newly found value lead to the establishment of a marketplace for software in the 1970s; more importantly, the newly found value encouraged the safe-guarding of source code as a means of protecting trade secrets, which was detrimental to the early version of FLOSS development (Glass, 2004).

Dissatisfied with the state of software development, Richard Stallman, a participant of MIT's Artificial Intelligence lab, established the Free Software Foundation (FSF). The goal of the organization was to promote the development and use of free software. The idea behind free software is that users are guaranteed the rights to freely use, distribute, and modify the software. The foundation contributed to the effort to develop free software solutions to many networking and software problems, such as Apache and Linux, which later powered the growth of the Internet. The Internet and free software's role in

[3] Adapted from our tutorial in CAIS (AlMarzouq et al., 2005)

its development further added to the success of the free software movement (Raymond, 2000a).

The success of free software, especially that coming from the distributed and open development model, inspired Eric Raymond to write his seminal piece *The Cathedral and The Bazaar*. His work played a key role in justifying the decision of Netscapeâ™ 's CEO, Jim Barksdale, to release the source code for Netscape Navigator in 1998. What followed was the recognition of the importance of marketing the free-software movement to ensure its long-term survival. This recognition eventually led to the term "Open Source" being coined as a less ambiguous term than "Free" and without the negative connotations in the business world. The Open Source Initiative (OSI) was then established to promote the ideals of Open Source to the business world (Raymond, 2000b).

The Open Source movement recognized the value of making source code accessible as a means to create higher quality software and was more pragmatic than the FSF in recognizing that the interests of the business world has to be met. Richard Stallman did not think the term Open Source conveyed the ideals of the FSF: that users had the right to freely modify, use, and redistribute the software (Raymond, 2000b). Therefore, the FSF and the OSI are two initiatives that promoted the same software development principles but differed in term of their ultimate goals (AlMarzouq et al., 2005).

One of the duties that both the FSF and OSI perform was the approval of FLOSS licenses that adhered to their ideals. Even today, the license is what distinguishes a software as being FLOSS (AlMarzouq et al., 2005). While the overall premise of every license must adhere to the principles of giving users the ability to use, modify, copy, and

redistribute software as they see fit, some licenses go beyond that principle by preventing the software's code from being mixed with proprietary source code and/or preventing the privatization of the source code (Stewart et al., 2006).

The software, however, is not the only valuable component of FLOSS. The community that builds such software is equally valuable. A FLOSS community consists of all of the users and developers of the software who are dispersed over time and space. Community members interact over the Internet and contribute source code, requirements, support for other members, and bug reports (Raymond, 2001; Scacchi, 2002). A community begins when a developer provides a proof-of-concept implementation of a software that members can collaboratively work on improving. As the software improve in features and quality, it initiates a self-enforcing growth cycle and attracts more users, and developers that contribute further to its development (Raymond, 2001). Most community members only contribute temporarily until their needs are satisfied, and only a small fraction of the members stay indefinitely with the community (Shah, 2006). Therefore, the survival of a FLOSS community will depend on its ability to continually attract new contributors to the development process.

1.1 The FLOSS Development Process

One could argue that the way in which the software development process changed with the emergence of FLOSS is the main reason why FLOSS became a significant force that changed the software industry. With traditional software development methods, an organization would assume all of the risks associated with software development and,

therefore, reap the rewards when the software is successful. The traditional software development method would start with requirements analysis to determine what features to incorporate into the software that will be built. Developers are then employed to implement and test the software. Finally, the organization building the software would continue to maintain the software after it has been released. The organization would assume the risks associated with failure in all of these steps in the development process, ranging from misspecification of the requirements and budget estimates to poor implementation and design. Because of the high risks associated with software development, it comes as no surprise that the majority of software projects fail (Brooks, 1975). Given these high risks, the rewards from the projects that do succeed result in the most profitable organizations in free-market history.

The FLOSS development process did away with much of the risks associated with the traditional software development process, mostly by doing away with deadlines and by distributing the risk amongst a larger group of stakeholders (i.e., contributors) and allowing them to pool their resources (Raymond, 2001). The availability of the source code allows users to modify the software for their own use, which leads to the discovery of hidden bugs, since these unique use cases often trigger unusual execution paths in the source code.

The increased number of eyes looking at the source code also reduces the likelihood that bugs will go undetected for a long time and increases the chances that someone will write a good fix from which all members will benefit (Raymond, 2001). The result of this collaborative behavior in the community is a reduction in the risks associated with im-

plementation, which results in higher quality software. In addition, the users are a source of ideas and inspiration for the developers. Since FLOSS community requirements are not set in stone like in traditional software-development settings, users can work on implementing the features that they would like to see, resulting in software that is continually improving in both its functionality and quality. We highlight further differences between traditional software and FLOSS development processes in Table 1.1.

Table 1.1: Comparison of conventional and FLOSS development process

Development Process	Conventional	FLOSS
Membership (Raymond, 2001)	Limited to project team	Open to any user of the software
Deadlines (Raymond, 2001)	Negotiate with customers and enforceable contracts	Release early and often and no enforceable deadlines
Roles (Crowston et al., 2005)	Defined by project manager	Self-selection based on need and interest
Leadership (Scozzi, 2008)	Explicit with role of project manager	Implicit and difficult to identify at times
Commitment of resources (van Wendel De Joode, 2004)	Developers are committed prior to engaging in development effort	Patches are contributed after work is completed by a developer, no requirement for commitment prior to engaging with a problem.
Requirements gathering (Scacchi, 2002)	Requirements made explicit.	Features emerge from community discourse

2 The Obsolescence of Conventional Software Development Wisdom

Based on our description of the FLOSS development process, we find that the benefits of FLOSS are a result of a greater number of participants in the development process. As more developers pool their resources, the less costly the development will be for any one developer and the more the risk of failure will be distributed. The discovery of bugs and the number of features suggested or implemented will also depend on the number of users and developers involved in the development process. Eric Raymond eloquently summarized the strength of FLOSS in what he dubbed, the Linus' Law: "Given enough eyeballs, all bugs are shallow" (p.19 Raymond, 2001)

The Linus' Law forwarded by Raymond seems to be at odds with conventional software engineering wisdom drawn from Brooks' Law. Brooks' Law can be summarized as "Adding developers to a late project only makes it later" (p. 25 Brooks, 1975). What Brooks' Law suggests, is that adding developers to an ongoing project is detrimental to its progress (Brooks, 1975). Brooks (1975) attributes the detrimental effect of adding developers to a software project to what he refers to as the ramp-up effect and to the communication effort required in large software development teams. We offer a pictorial summary of Brooks' Law in Figure 1.1.

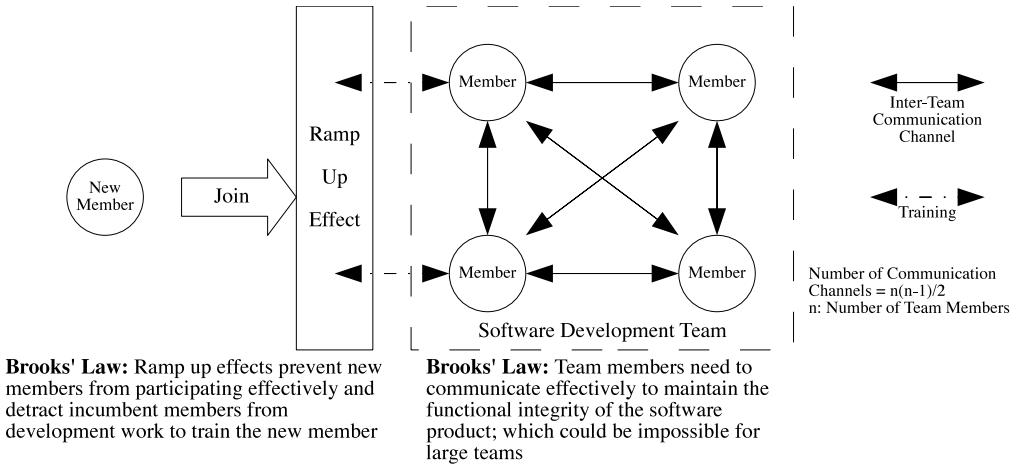

Figure 1.1: Pictorial depiction of Brooks' Law.

The ramp-up effect refers to the training effort required to get new members up to speed, such that they become productive team members. The duty of training these new members will fall on productive incumbent members. As a result, the productivity of these incumbent members will decrease, as they are distracted from working on the project (Brooks, 1975).

Furthermore, adding more people to a project will increase the number of communication channels that any developer needs to maintain in order to ensure the functional integrity of the software project (Brooks, 1975). Two developers will be required to expend less effort communicating about their work to one another than 20 members would. Each developer of the 20-member group will need to communicate to all of the other 19 members, which takes a significant portion of the members' time and hurts overall team productivity.

Raymond (2001) went as far as declaring Brooks' Law obsolete in the FLOSS context. He attributes the differences of the FLOSS development process in addition to the ability to use source code as a communication medium, a fact that Brooks (1975) overlooked, as being the main reasons why Brooks' Law no longer applies. For example, the release-early/release-often mentality employed in FLOSS implies that contributions from external contributors are incremental in nature. Provided that the source code is solid and the patch is well written, patches can be easily comprehended and incorporated into the main code base with minimal effort. As such, new contributors will not place a significant learning or communication burden on the current development team in a FLOSS community.

3 The Research Question

While Raymond's (2001) arguments) are compelling and have even found empirical support (e.g. Koch, 2004; Schweik et al., 2008), we believe that recent changes in the FLOSS movement (Fitzgerald, 2006) warrant a revisit to the conflict between Raymond's and Brooks' views.

FLOSS has had a surge in interest in recent years and has been increasingly adopted by for-profit organizations (Fitzgerald, 2006), which invalidates the view that FLOSS contributors are mostly hobbyists (Shah, 2006; Raymond, 2001). The 2007 report by the Linux Software Foundation found that the majority of contributors to the Linux Kernel were employed by the largest organizations in the software and hardware markets (Kroah-Hartman et al., 2008). Furthermore, researchers observed that some of the most

successful projects are able to maintain super-linear growth of the source code (Herraiz et al., 2006). Therefore, it is possible that the assumptions held by Raymond (2001) might differ from the current realities of FLOSS development efforts.

Indeed, researchers have begun to question many of Raymond's (2001) assumptions and descriptions. For example, not all FLOSS communities observed a distributed bazaar-like development model, as the majority of FLOSS communities are highly centralized (Krishnamurthy, 2002; Crowston et al., 2005). In addition, not all communities seem to follow the release-early/release-often mentality (Krishnamurthy, 2002; Herraiz et al., 2006). More recently, there has even been work suggesting that Brooks' Law might not be obsolete after all (Capiluppi and Adams, 2009). Therefore, it seems like there is mounting evidence that we still do not have a complete understanding of the FLOSS development process in its more recent reincarnation.

The goal of this work is to resolve the paradox in our understanding of the software development process in the FLOSS context. Conventional wisdom points to the detrimental effect of team size in a software development team, while Eric Raymond's description of the FLOSS development process and recent empirical work suggests that size is beneficial. What is at stake here is the sustainability of the FLOSS development process and the livelihood of FLOSS communities. Failure to understand the implications of Brooks' Law in the FLOSS context might result in dysfunctional development processes. Therefore, with this work, we will attempt to answer the following research question: "Is Brooks' Law obsolete in the context of FLOSS development?"

To address this general research question, we need to take into consideration the differences between the conventional and FLOSS development processes (see Table 1.1). There are a number of differences between the two processes that require us to reexamine the assumptions behind Brook's Law before we declare it obsolete or not: namely, the voluntary nature of participation in FLOSS and the absence of enforceable deadlines in FLOSS development. We will also attempt to address issues that have been ignored by literature examining Brooks' Law in the FLOSS context: mainly, the nature and effect of the ramp-up effect (Raymond, 2001; Koch, 2004; Schweik et al., 2008; Capiluppi and Adams, 2009). As such, we will perform two separate studies that address the main aspects of Brooks' Law. The first will be concerned with the nature of the ramp-up effect and how it impacts a FLOSS community. The second will directly address the tension between Brooks' and Raymond's views with regards to the effect of greater numbers in FLOSS development communities.

Answering our research question will be of practical and theoretical importance. We have highlighted the role FLOSS plays in enabling new service-based business models, such as cloud computing services. Understanding the nature of the FLOSS development process will be important to any company that seeks to build a business model that relies on FLOSS, as these companies will have to engage the FLOSS community in order to effectively build their services. Knowing how the FLOSS development process works allows such companies to participate in the most effective manner. This understanding of the FLOSS development process will be equally important to FLOSS community managers, as this understanding would empower them to shape the development process more

effectively and make it more efficient in order to utilize the time offered by contributing volunteers contributing in the most beneficial way possible.

There are also important theoretical implications to this work, such as understanding the boundary conditions of Brooks' Law and consolidating two important but conflicting views of the dynamics of the FLOSS development process. Furthermore, our work will lead us to understand the nature and implications of the ramp-up effects in FLOSS communities more fully. Finally, the results of this work will allow us to understand the factors that contribute to the increased performance of the software development process.

3.1 The Studies

To address our general research question, we will perform two separate studies that address different aspects of Brooks' Law and how it might apply to the FLOSS development context (see Figure 1.2 for an overview of the studies). The first of these studies will be concerned with the nature of the ramp-up effect and its impact on FLOSS communities. The second study is concerned with the impact of developer numbers on the performance of FLOSS communities.

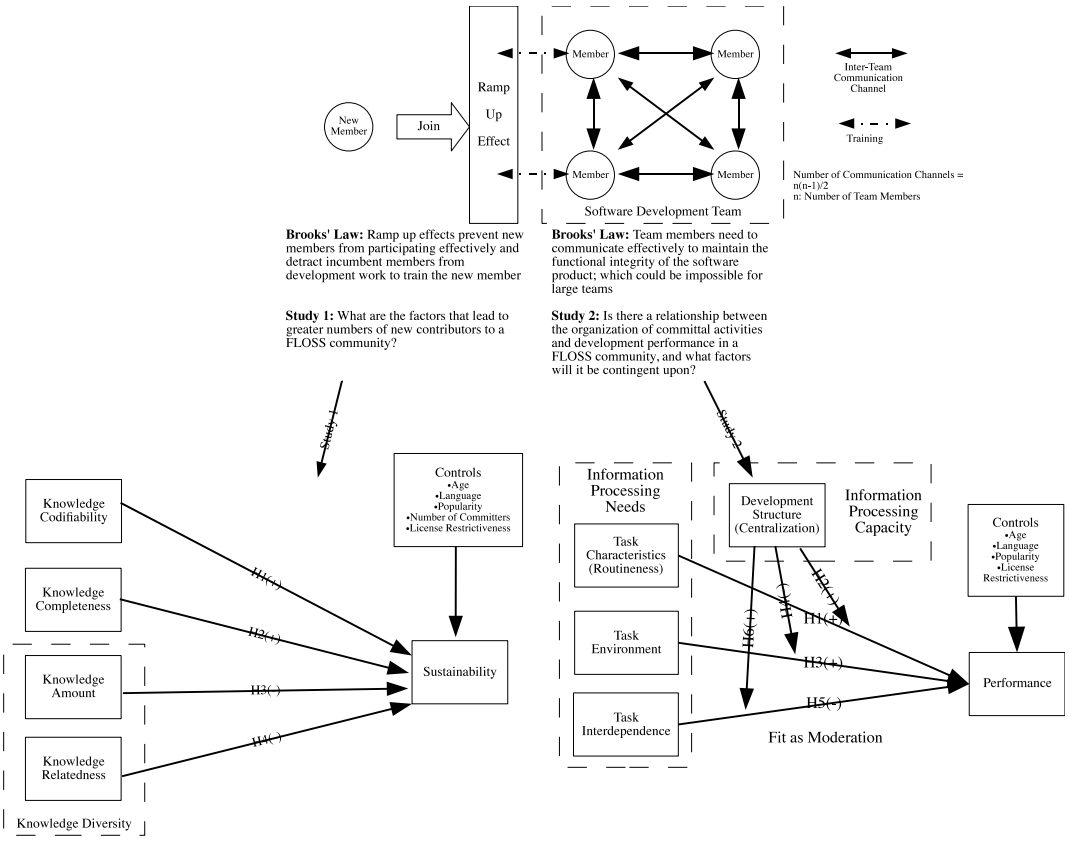

Figure 1.2: Position of studies with respect to Brooks' Law

3.1.1 Study 1

According to Brooks (1975), the ramp-up effect will reduce the productivity of a software development team because current members are responsible for training new members that join the development team after it has begun. In FLOSS development, current developers are under no obligation to train any new members, so how can we conceptualize of the ramp-up effect?

We conceptualize the ramp-up effect in this study as knowledge barriers that incumbent members of conventional software development teams help new members overcome through training (Attewell, 1992). While it is possible for current developers in a FLOSS

to also help new contributors, they are under no obligation to do so, especially since the source code can convey all of the knowledge required to become a developer (Raymond, 2001). Therefore, assuming that the source code is the main knowledge source for new contributors, the ramp-up effect could vary from community to community based on the characteristics of the source code.

With this conceptualization of the ramp-up effect, we view source code design as a kind of resistor to the participation of new contributors. When the resistance is high, potential contributors are required to expend considerable effort to acquire the necessary knowledge to become contributors. Since most contributors are volunteers, having a high level of resistance from the source code (i.e., knowledge barrier) would result in fewer new contributors for projects relative to projects with less resistance. As such, we ask "what are the factors that lead to greater numbers of new contributors in a FLOSS community?"

To answer this question, we conceptualize FLOSS communities as being competitive actors in what we refer to as the FLOSS marketplace. In this FLOSS marketplace, communities compete to gain contributions from developers in order to sustain their development activity. We also conceptualize the act of contribution as a transaction that is completed once a source code contribution is fully incorporated into the community code base. Using the insights from Transaction Cost Theory (TCT) (Coase, 1937; Williamson, 1975), we identify the characteristics of the underlying development knowledge as the main factors that increase the costs of contribution (i.e., increased knowledge barriers). Specifically, we argue that FLOSS communities could increase the numbers of new con-

tributors if they could manage the codifiability, completeness, and diversity of their underlying development knowledge.

3.1.2 Study 2

In this study, we directly address the tension between Brooks' view and Raymond's view on developer numbers in FLOSS development teams. According to Brooks (1975), a high number of developers is detrimental to the progress of a software development team because of the effort required to maintain open communication channels by any one developer with the other members of the team. With more communication channels, the effort becomes significant and can distract the developer from his/her main development work, thereby, reducing the overall productivity of the team.

Raymond (2001), on the other hand, views larger numbers as beneficial for FLOSS development teams because the result is higher quality software and more productive communities. He argues against the need for significant communication efforts in FLOSS teams because the source code itself could be a rich communication medium.

In this study, we identify what we refer to as the committal structure of the FLOSS community and suggest that it is a potential bottleneck for the FLOSS development process (Goldratt and Cox, 1994). The committal structure represents the way in which committal activities are performed (i.e., how patches are integrated into the code base). This study differs from the first study in that it focuses on the efforts shouldered by the committers during committal activities and how it might impact the performance of the community. In the first study, we focus on the effort shouldered by potential contributors up to

the act of committal and how that process may impact the contributors' decision to go through with the process. Therefore, we identify the main research question for this study as "Is there a relationship between the organization of committal activities and development performance in a FLOSS community, and upon what factors will it be contingent upon?"

To address this question, we conceptualize FLOSS communities as information-processing systems in which the information-processing task is the act of committal. Using insights from Organizational Information Processing Theory (OIPT) (Galbraith, 1973), we argue that the performance of FLOSS communities is contingent upon the capacity of the committal structure to process information and the information-processing requirements of the committal task. Using this conceptualization, we also reach an understanding of how performance can be viewed for FLOSS communities in which deadlines are not relevant. We identify task routineness, contributor uncertainty, and task interdependence as the main sources of uncertainty that increase the information-processing requirements of a FLOSS development community and, therefore, the performance of the community.

4 Literature Review

4.1 Study 1

We summarize the literature pertaining to why developers join FLOSS development efforts in Table 1.2. What the table makes apparent is the following:

- All of the literature in the table focuses on motivational factors as opposed to factors that might cause resistance to participation with the exception of the study by Midha (2008), which lacks an overarching theoretical modlel and might be considered exploratory in nature.
- Most of the literature focuses on individual characteristics with the exception of Baldwin and Clark (2006), which is purely theoretical; MacCormack et al., (2006), which is a case study with limited external validity; and Midha (2008), which lacks a strong theoretical foundation.
- Empirical studies in the table have limited generalizability because of the research approach (i.e., case study) or use sourceforge.net samples that can no longer be considered be representative of the FLOSS population (Paul, 2009b, 2009a).
- The tools used in studies that examine the structure of the code are somewhat complicated and do not offer the means to feasibly compare the designs of multiple code bases (e.g., MacCormack et al., 2006)

In our effort to address our first study's research question, we hope to also address the gaps that we have highlighted in Table 1.2. Specifically, we will give a resistance- (i.e., cost) based explanation as to why developers contribute to FLOSS projects. Our work will also be of an empirical nature, focusing the project as the main unit of analysis. We will utilize a method proposed by Newman (2006a) to examine the structures of networks as a means to examine, assess, and compare the designs of FLOSS code bases and empirically test the idea of structures of participation (Baldwin and Clark, 2006). Finally, we

will test our model using archival data, which we expect to result in findings with greater external validity than the previously conducted empirical studies on FLOSS participation.

Table 1.2: FLOSS studies related to new member participation

Study	Type	LoA	Relevant Findings
(Lerner and Tirole, 2002)	Theoretical	Individual	Motivation was derived from indirect signaling about quality, with the payoff to come in higher career earnings.
(Hars and Ou, 2001)	Empirical - Survey of 81 FLOSS participants	Individual	Individual motivations to contribute to FLOSS can be classified into intrinsic and extrinsic categories.
(Ye and Kishida, 2003)	Theoretical - Based on learning theory of legitimate peripheral participation	Individual	FLOSS contributors are motivated by learning, which is enabled by the availability and modularity of the source code.
(Lakhani and von Hippel, 2003)	Empirical - Survey and archival data of 336 Apache participants	Individual	Mundain but necessary tasks are performed by participants for their own learning benefit.
(von Krogh et al., 2003)	Inductive Theory - Interviews and archival data from Freenet project	Individual	Individuals gain committal privileges through technical and constant contributions.
(Hann et al., 2004)	Empirical - Survey of 122 contributors to three Apache projects	Individual	Use value, recreational value, and potential career impact were found to be the main motivators for participation in the sample.
(Roberts et al., 2006)	Empirical - Survey and Archival data of 288 Apache contributors.	Individual	Intrinsic and extrinsic motivations for participation are related and have an impact on individual performance.
(MacCormack et al., 2006)	Exploratory study - Comparison of Linux and Mozilla code structure	FLOSS Project	FLOSS project development is enabled by code structures that enable participation (i.e., modular).
(Baldwin and Clark, 2006)	Theoretical - Modeling	FLOSS project	Modularity and option value of modules encourage participation.
(Shah, 2006)	Inductive theory - Two communities from sourceforge.net	Individual	Need is the biggest driver of participation, most contributors leave the community then. A small subset continues with the community and contribution becomes more of a hobby.
(Midha, 2008)	Empirical - Archival data from 450 C/C++ based sourceforge.net projects	FLOSS project	Change in aggregate McCabe's cyclomatic complexity measure was found to be related to the number of new committers.

4.2 Study 2

While the empirical work on the performance of FLOSS communities is more diverse than participation, we notice that the notions of performance and goals are also diverse

and are usually referred to as success. We summarize the main studies that we examined for this study in Table 1.3, which highlights the following gaps in the FLOSS performance literature:

- With the exception of Capiluppi and Adams (2009), all of the studies seem to support the obsolescence of Brooks' Law directly or indirectly.
- There is no treatment of the relationship between the structure of the source code and the development structure in the listed studies.
- Studies taking a social-network-analysis approach make the assumption that performance is a result of some of the communication patterns observed in the communication structure. However, there are a number coexisting structures present in a FLOSS community, such as the development and communication structures (Mockus et al., 2002). No clear theoretical underpinning is given as to why performance is caused by the structures examined in these studies.

With this study, we hope to find the answer to the main research question and, in the process, address the highlighted limitations in the current literature. We hope to specifically address the tension between Brooks' view and Raymond's view of FLOSS development and explain why there are conflicting results in the literature. We will reach that result by using the insights from Organizational Information Processing Theory (OIPT) (Galbraith, 1973) in order to understand the relationship between the structure of software and the development structure and how their fit could relate to performance. More importantly, we identify the committal structure, consisting of committers, as the main

development bottleneck that could limit performance (Goldratt and Cox, 1994). With this theory-driven approach, we hope to avoid the main concerns that have caused doubt about the internal validity of many prior studies on FLOSS performance.

Table 1.3: FLOSS studies related to development productivity

Study	Type	LoA	Relevant Findings
(Mockus et al., 2002)	Case Study - Apache web server and Mozilla Browser communities with follow-up quantitative study on Mozilla archival data	Multiple	FLOSS communities use different coordination mechanisms depending on the size of the development team. In addition, the relative size of the core developer group to other groups in the FLOSS community will have different implications on the productivity and sustainability of the community in addition to the software quality.
(von Krogh et al., 2003)	Inductive Theory - Interviews and archival data from Freenet project	Individual	Committers specialize in a specific component when they are first granted committal privileges due to contribution barriers.
(Sagers, 2004)	Empirical - Survey and archival data from 38 FLOSS projects	FLOSS project	Using network governance theory, success was found to be related to social mechanisms of coordination and safeguarding. These mechanisms are restricted in terms of access to the source code, collective sanctions, and the reputation of developers. Success was measured as self-reported performance of the project, age of the project, and ratio of open issues (i.e., bug report and feature requests) to the total number of issues.
(Long and Yuan, 2005)	Empirical - Archival data of 300 FLOSS projects hosted on sourceforge.net	FLOSS Project	Success, in terms of number of downloads, is closely related to core developer advocacy.
(Daniel et al., 2006)	Empirical - Archival data of 78 FLOSS projects hosted on	FLOSS Project	The user and developer groups in a FLOSS project enhance the absorptive capacity of a FLOSS project through two salient capabilities: knowledge transfer and knowledge acquisition. FLOSS project with higher absorptive capacities were found to perform better in terms of closed tickets and lines of code added.
(Grewal et al., 2006)	Empirical - Archival data of 108 FLOSS projects onsourceforge.net	FLOSS Project	The embeddeness of a network (i.e., how interconnected a project is to other projects through its members) is more closely related to technical success than commercial success. Technical success was measured as the number of commits, while commercial success as the number of downloads.
(Liu and Iyer, 2007)	Empirical - Archival data of 200 FLOSS projects hosted on sourceforge.net	FLOSS Project	The software design and communication structure of the developers will be related to success in terms of the quality of the developed software and the velocity of the development.

(Singh, 2007)	Empirical - Archival data of 2013 FLOSS projects hosted on sourceforge.net	FLOSS Project	Small world properties (i.e., dense clustering and short average patch of communication between developers) are found to be positively related to productivity in terms of number of commits.
(Tan et al., 2007)	Empirical - Archival data of 5191 FLOSS projects hosted on sourceforge.net	FLOSS Project	Direct and indirect ties, in addition to communication network cohesion, were all found to be positively related to the productivity of the FLOSS community as measured by number of commits.
(Wu et al., 2007)	Empirical - Archival data of 59 FLOSS projects hosted on sourceforge.net	FLOSS project	Project characteristics (e.g.., license and complexity) and communication patterns (i.e., centrality and density) are related to the performance of the FLOSS project.
(Midha, 2008)	Empirical - Archival data from 450 C/C++ based sourceforge.net projects	FLOSS project	Change in aggregate McCabe's cyclomatic complexity measure was found to be related to the number of bugs and the time to fix bugs.
(Capiluppi and Adams, 2009)	Empirical - Archival data from KDE project repository	Individual	The average number of communication channels generally decreases with increased numbers of contributors but increases significantly after a certain threshold number of concurrent developers was reached.

5 Results and Implications

Following the call of Koch (2004) to perform empirical research with more external validity, we empirically test our research models using data that we believe is more representative of the FLOSS population than prior studies. While prior studies relied mainly on data collected from sourceforge.net, which hosts FLOSS projects (e.g., Stewart et al., 2006, 2006; Koch, 2004; Liu and Iyer, 2007; Midha, 2008), we selected our sample from ohloh.net, which lists FLOSS projects regardless where or how they are hosted.

We believe our sample to be more representative given the recent decline of sourceforge.net as the leading hosting website for FLOSS projects (Paul, 2009b, 2009a). In addition, we found that only 22% of the top projects listed on ohloh.net (ordered by popularity) to be hosted on sourceforge.net.

Using archival data from the source code repositories of 234 FLOSS projects, we collect quarterly observations for the relevant variables in both of our studies between the years 2007 and 2009 for a total of 1823 observations. Given the longitudinal nature of our observations and variations in the number of observations per project, we used mixed model analysis (Cohen et al., 2003) to fit our statistical model and make inferences about our observations. We summarize the main findings and implications of the results in the following sections.

5.1 Study 1

The importance of our first study stems from the conceptualization of the ramp-up effect in the FLOSS context and how it relates to the sustainability of a FLOSS community and the relationship between new-member participation and the sustainability of the development effort in a FLOSS community. Using Transaction Cost Theory (TCT) (Coase, 1937; Williamson, 1975), we identify knowledge codifiability, knowledge completeness, and knowledge diversity as the main dimensions of the underlying development knowledge that relates to the numbers of new contributors.

Table 1.4: Summary of Study 1's main findings

Finding	Impact
FLOSS marketplace conceptualization.	• Delineate the differences between the software and the FLOSS marketplaces. • Application of theories related to market competition to understand key aspects of FLOSS.
Knowledge codifiability is positively related to an increase in the number of new contributors.	• While source code is a codified form of knowledge, there are also tacit assumptions that can be made explicit in code documentation. • Importance of good code documentation.
Knowledge completeness is positively related to an increase in the number of new contributors.	• While software development is an inherently uncertain task, this uncertainty tends to vary between FLOSS projects. • Highly modularized projects reduce the uncertainty inherent in the software development task and could result in greater participation in FLOSS.
Knowledge diversity is negatively related to an increase in the number of new contributors.	• Amount and relatedness of knowledge are two distinct dimensions of knowledge diversity with their own unique impact on participation. • Projects with highly diverse underlying knowledge might become more manageable if broken down into smaller FLOSS projects.
The impact of knowledge codifiability and diversity on the numbers of new contributors follows a diminishing returns pattern.	• FLOSS communities need to play a balancing act in terms of how much of the code base should be documented and must always keep the documentation up to date. • FLOSS communities need to be clear on the goals of their projects and be able to say no to the inclusion of certain features such that their projects can remain manageable.
Application of the leading eigenvector and modularity measure (Newman, 2006a) to understand the structure of source code.	• Objective measure of modularity that is programming-language-agnostic. • Software engineering tool that could assist in assessing code reorganization efforts.

The findings from the first study have some important theoretical implications. In developing our theory, we framed the relationship between developers and FLOSS communities as that of a marketplace in which the ramp-up effect is seen as a knowledge barrier and a source of transaction cost. This framing opens up the possibility of studying FLOSS communities in light of strategic organizational theories to understand how communities can be more competitive in the marketplace. In addition, our work gives us a better understanding of the relationship between the characteristics of development-related knowledge and the number of new contributors joining the development effort in a FLOSS community.

Finally, we contribute by introducing a novel measure of modularity suggested by Newman (2006b), which will be important to understand the effects of code organization more fully. The empirical results in this study suggest that there is validity to this method. What is unique about this measure is that it is language-agnostic, as it relies mostly on the dependencies between source code files to estimate modularity. Such a property would make the measure not only useful for larger groups of developers using different programming languages but also for theorists who wish to get at the heart of the concept of modularity without being influenced by programming–language-specific constructs.

Our findings also have some important practical implications for FLOSS community organizers. The findings suggest that communities that manage the ramp-up effect by reducing contribution costs will be able to attract more contributors and increase their chances sustaining their development effort for a longer period of time. Such costs are

manageable if the FLOSS community dedicates effort to tasks that are usually considered mundane, unexciting, and likely to attract the least interest from contributors, such as documentation and code reorganization. However, the results also suggest that there is a limit to managing such costs given the tendency of a source code to grow in complexity (Lehman and Belady, 1985)In addition, the modularity measure has the potential to be used as a tool by developers to understand the effect of their code reorganization efforts and whether such efforts actually improve the code structure or not.

5.2 Study 2

For the second study, we directly addressed the tension between Brooks' and Raymond's views of software development. Based on Information Processing Theory (Galbraith, 1973) and the assumption that committers are boundedly rational (Simon, 1955), we identify the committal structure as the performance bottleneck in a FLOSS community. We summarize the main findings and their implications in Table 1.5

Our findings have important theoretical implications for research on FLOSS productivity and community organization. We suggest that no committal structure is a superior choice; rather, it is the communities' conditions that should dictate which structure to employ. We identify task routineness, contributor uncertainty, and task interdependence as the main factors that influence a community's decision to choose the optimal committal structure.

Table 1.5: Summary of Study 2's main findings

Finding	Impact
FLOSS communities performing simple tasks perform better.	• Importance of good source code design to simplify development. • Contributors should work closer to the community and make small incremental changes rather than work in isolation and accumulate their patches into a single patch that is hard to incorporate.
Centralized committal structures are a better fit for routine tasks.	• The committal structure should match the needs of the community. • There is no single superior structure.
Decentralized committal structures are a better fit under high contributor uncertainty.	• Decentralized committal structures are necessary if community involvement is valued. • Brooks' Law is not obsolete, but the committal structure has to be overwhelmed for it to become obvious.
Task interdependence increases information-processing requirements on a FLOSS community and reduces performance.	• Importance of modularizing the source code and its effect on the performance of a FLOSS community. • Further validation of the Newman (2006a) modularity measure.
Decentralized committal structures are a better fit under conditions of low task interdependence.	• Decentralized committal structures are enabled by proper code design. • Centralized committal structures might be the only way to continue to maintain and develop tightly coupled code bases. • Brooks' and Raymond's views are complementary. Raymond explains how FLOSS development is conducted under conditions of fit, while Brooks' views become apparent under condition of lack of fit.

The findings also suggest that FLOSS development teams are no different from any other development team and that that our understanding of managing FLOSS communities could benefit greatly from the organizational body of knowledge. Using OIPT, we elaborate on the concept of fit to for a community to achieve superior performance. Fit requires that the committal structure, as the source of information-processing capacity, should match the requirements of the development task. With lack of fit, a FLOSS community could either work on improving their information-processing capacity through the use of development tools and a more decentralized committal structure, or they could reduce their information-processing needs through code reorganization efforts.

Finally, and most importantly, the findings suggest that the views of Brooks' and Raymond are not mutually exclusive; rather they suggest that having more contributors is indeed beneficial to the performance of FLOSS communities under the condition that the community achieves a fit between its committal structure and the information-processing requirements of the committal task. When there is a lack of fit, communities start to experience a degradation in performance or have no performance gains as the number of contributors increases.

The findings also have important practical implications ,as they provide a better understanding for community organizers in terms of how to put less burdens on committers by properly organizing them. This, in turn, reduces the risk of losing them after a short time of service. Furthermore, these results give for-profit organizations an understanding of the similarities between FLOSS development teams and traditional software development teams, thereby leveraging their current knowledge base to improve the de-

velopment performance of FLOSS communities further, especially since deadlines are a great concern for many of the for-profit organizations involved with FLOSS.

6 Conclusion

The two studies we conducted combine to give a detailed picture of the tension between Brooks' and Raymond's views of software development. In the first study, we forward a conceptualization of the ramp-up effect as knowledge barriers that impact the sustainability, and potentially performance, of FLOSS communities by limiting the number of new contributors that join the development effort. The second study complements this understanding by directly exploring the interplay between a community's structure and the source code's structure and how it impacts the community's overall performance. We also explain, in this last study, how performance in FLOSS communities should be conceptualized relative to other communities due to the irrelevance of deadlines in the FLOSS context.

Based on these two studies, we conclude that Brooks' and Raymond's' views are not at odds but actually complement one another. We found that Raymond's views hold when the committal structure is not overwhelmed. In such conditions, the community can handle an increase in the numbers of contributors and could actually benefit from their contributions and increase performance. However, in conditions of lack of fit in which the committal structure is overwhelmed, greater numbers could be detrimental to the performance of the community, as committers have to make a tradeoff between their own development work and committing the work of others. It is in such conditions that

Brooks' view becomes prevalent in FLOSS. As such, we attribute the conflicting results in research to the differing contexts and assumptions in these studies.

References

AlMarzouq, M., Zheng, L., Rong, G., and Grover, V. (2005). Open source: Concepts, benefits, and challenges. *Communications of AIS*, 2005(16):756–784.

Attewell, P. (1992). Technology diffusion and organizational learning: The case of business computing. *Organization Science*, 3(1): –19.

Baldwin, C. Y. and Clark, K. B. (2006). The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116–1127.

Bonaccorsi, A., Giannangeli, S., and Rossi, C. (2006). Entry strategies under competing standards: Hybrid business models in the open source software industry. *Management Science*, 52(7):1085–1098.

Brooks, F. (1975). The mythical man-month. In *Proceedings of the International Conference on Reliable Software*, volume 10. ACM Press.

Capiluppi, A. and Adams, P. J. (2009). Reassessing Brooks' Law for the free software community. In Boldyreff, C., Crowston, K., Lundell, B., and Wasserman, A. I., editors, *OSS*, volume 299 of *IFIP*, pages 274–283. Springer.

Coase, R. H. (1937). The nature of the firm. *Economica*, 4(16):386–405. ArticleType: primary_article/Full publication date: Nov., 1937 / Copyright © 1937 The London School of Economics and Political Science.

Cohen, J., Cohen, P., West, S., and Aiken, L. (2003). *Applied multiple regression/ correlation analysis for the behavioral sciences*. Lawrence Erlbaum, third edition.

Crowston, K., Wei, K., Li, Q., Eseryel, U., and Howison, J. (2005). Coordination of free/ libre open source software development. In *ICIS 2005 Proceedings*.

Daniel, S., Agarwal, R., and Stewart, K. (2006). An absorptive capacity perspective of open source software development group performance. In *ICIS 2006 Proceedings*.

Economides, N. and Katsamakas, E. (2006). Two-sided competition of proprietary vs. open source technology platforms and the implications for the software industry. *Management Science*, 52(7):1057–1071.

Fitzgerald, B. (2006). The transformation of open source software. *MIS Quarterly*, 30(3):587–598.

Galbraith, J. R. (1973). *Designing Complex Organizations*. Addison-Wesley series on organization development. Addison Wesley.

Glass, R. L. (2004). A look at the economics of open source. *Commun. ACM*, 47(2):25–27.

Goldratt, E. M. and Cox, J. (1994). *The Goal*. North River Press, second edition.

Grewal, R., Lilien, G. L., and Mallapragada, G. (2006). Location, location, location: How network embeddedness affects project success in open source systems. *Management Science*, 52(7):1043–1056.

Hann, I.-H., Roberts, J., and Slaughter, S. (2004). Why developers participate in open source software projects: An empirical investigation. In *ICIS 2004 Proceedings*.

Hars, A. and Ou, S. (2001). Working for free?: Motivations of participating in open source projects. In *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 7*. IEEE Computer Society.

Herraiz, I., Robles, G., and Gonzalez-Barahon, J. M. (2006). Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 206–213, Washington, DC, USA. IEEE Computer Society.

Koch, S. (2004). Profiling an open source project ecology and its programmers. *Electronic Markets*, 14(2):77–88.

Krishnamurthy, S. (2002). Cave or community?: An empirical examination of 100 mature open source projects. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/960/881.

Kroah-Hartman, G., Corbet, J., and McPherson, A. (2008). Linux Kernel development (April 2008). https://www.linuxfoundation.org/publications/linuxKerneldevelopment.php.

Lakhani, K. R. and von Hippel, E. (2003). How open source software works: "free" user-to-user assistance. *Research Policy*, 32(6):923–943.

Lehman, M. M. and Belady, L. A., editors (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA.

Lerner, J. and Tirole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics*, 50(2):197.

Liu, X. and Iyer, B. (2007). Design architecture, developer networks and performance of open source software projects. In *ICIS 2007 Proceedings*.

Long, J. and Yuan, M. J. (2005). Are all open source projects created equal?: Understanding the sustainability of open source software development model. In *AMCIS 2005 Proceedings*.

MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015–1030.

Midha, V. (2008). Does complexity matter?: The impact of change in structural complexity on software maintenance and new developers' contributions in open source software. In *ICIS 2008 Proceedings*.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309–346.

Newman, M. E. J. (2006a). Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74:036104.

Newman, M. E. J. (2006b). Modularity and community structure in networks. *PNAS*, 103:8577.

Paul, R. (2009a). SourceForge adds support for new version control systems. http://arstechnica.com/open-source/news/2009/03/sourceforge-adds-support-for-new-version-control-systems.ars.

Paul, R. (2009b). SourceForge wants to be collaboration powerhouse, buys ohloh. http://arstechnica.com/open-source/news/2009/05/sourceforge-acquires-foss-code-metric-web-site-ohloh.ars.

Raymond, E. (2000a). A brief history of hackerdom. http://www.catb.org/~esr/writings/cathedral-bazaar/hacker-history/.

Raymond, E. (2000b). Revenge of the hackers. http://www.catb.org/~esr/writings/cathedral-bazaar/hacker-revenge/.

Raymond, E. (2001). *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary*. O'Reilly, Cambridge, MA, revised edition.

Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. *Computer*, 40(4):25–32.

Roberts, J. A., Hann, I., and Slaughter, S. A. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984–999.

Sagers, G. (2004). The influence of network governance factors on success in open source software development projects. In *ICIS 2004 Proceedings*.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *Software, IEE Proceedings*, 149(1):24–39.

Schwartz, R., Bacon, J., and Laporte, L. (2009). Floss weekly 73: Tim O'Reilly. Podcast.

Schweik, C. M., English, R. C., Kitsing, M., and Haire, S. (2008). Brooks' versus Linus' law: An empirical test of open source projects. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research*, pages 423–424. Digital Government Society of North America.

Scozzi, B., C. K. E. U. L. Q. (2008). Shared mental models among open source software developers. In *Hawai'i International Conference on System Science*.

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

Simon, H. A. (1955). A behavioral model of rational choice. *The Quarterly Journal of Economics*, 69(1):99–118.

Singh, P. (2007). Open source software development and the small world phenomenon: An empirical investigations of macro level colaboration network properties on project success. In *ICIS 2007 Proceedings*.

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.

Tan, Y., Mookerjee, V., and Singh, P. (2007). Social capital, structural holes and team composition: Collaborative networks of the open source software community. In *ICIS 2007 Proceedings*.

van Wendel De Joode, R. (2004). Managing conflicts in open source communities. *Electronic Markets*, 14(2):104–113.

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.

Williamson, O. E. (1975). *Markets and hierarchies: Analysis and antitrust implications*. The Free Press, New York, NY.

Wu, J., Goh, K.-Y., and Tang, Q. (2007). Investigating success of open source software projects: A social network perspective. In *ICIS 2007 Proceedings*.

Ye, Y. and Kishida, K. (2003). Toward an understanding of the motivation open source software developers. In *Proceedings of the 25th International Conference on Software Engineering*, pages 419–429, Portland, Oregon. IEEE Computer Society.

Chapter 2

Study 1: The FLOSS Marketplace and The Sustainability of FLOSS Communities

Abstract

A great deal of the Free/Libre and Open Source Software (FLOSS) literature inappropriately assumes that the reasons FLOSS community members make source code contributions are the same. The literature cites several reasons that members contribute, including satisfying one's own needs, demonstrating one's ability to potential employers, and gaining peer recognition; however, much of this literature does not take into account the differences between new and experienced contributors. We argue that these two groups of contributors are faced with different cost components related to contributing and that a FLOSS community's ability to nurture new contributors will be important for its long-term survival. To understand why new contributors participate, we distinguish between the software and FLOSS marketplaces and present the conditions that could influence rational profit-maximizing actors' decisions not to contribute to a FLOSS project. With this framing, we develop a model based on Transaction Cost Theory that sheds light on how FLOSS communities can effectively convert users of the software into members that contribute to its development. Based on this model, we conclude that the knowledge characteristics required to make a contribution are the main impediment for first-time contributors. We argue that improving software design is the most effective way for FLOSS communities to encourage new contributors to participate in the development effort, as it reduces the burden on potential contributors to acquire the knowledge necessary to make

source code contributions. We discuss the process by which we will test this model and our use of a novel approach to measuring software modularity. Finally, we conclude with a discussion of the implications and limitations of our study.

1 Introduction

Free, Libre, and Open Source Software (FLOSS) is licensed to enable users to redistribute and modify it freely (Raymond, 2001). As a result, users can easily obtain a copy of the code base and participate in development efforts by contributing their modifications to various FLOSS projects. These projects rely on such voluntary contributions for their continued development and growth (Lee and Cole, 2003).

Because users can obtain FLOSS at no cost, some individuals simply use these programs without contributing to their development. Known as the free-rider problem, this occurs when individuals take advantage of common-pool goods without taking action to support the community that develops them (von Hippel and von Krogh, 2003; O'Mahony, 2003; English and Schweik, 2007). Should free-riding behavior become prevalent, the long-term survival of FLOSS communities would be seriously endangered. Therefore, to understand FLOSS sustainability, it is extremely important to understand the factors that lead individuals to voluntarily contribute source code.

Individual contributions to FLOSS communities take many forms. Some contributors report bugs or suggest features, while others write code to improve the software (Mockus et al., 2002). In this study, we are primarily interested in code writers, as the FLOSS products would have not been produced without their efforts.

According to Shah (2006), code writers can be classified into two groups: contributors and committers. Contributors are those individuals considered external to the main devel-

opment efforts in the community and occasionally contribute patches of code. Committers are members that have the necessary privileges to approve and incorporate patches into a community code base. In addition, the committers perform the largest share of development and are, therefore, considered part of the FLOSS community's core development team (Crowston and Howison, 2005).

Although contributors and committers differ in terms of their level of involvement and the frequency of their contributions, the literature on FLOSS participation does not offer any clear, theory-driven distinctions between these types of participants in FLOSS communities. Within the FLOSS literature, there is an implicit assumption that all participants contribute to FLOSS communities for the same primary reasons. For example, the literature suggests that gaining social status motivates individual developers to participate in code development rather than simply free-ride (Raymond, 2001; Lakhani and Wolf, 2007). Alternately, individual participants may contribute to demonstrate their software-development abilities to potential employers (Lerner and Tirole, 2002; Roberts et al., 2006). Some would also suggest that individual developers contribute source code simply to satisfy their personal interests (Raymond, 2001).

What the FLOSS motivational literature overlooks is a barrier that impacts any new participant in any ongoing software development effort known as the ramp-up effect (Brooks, 1975). The ramp-up effect refers to the training and knowledge that new participants in a software development effort need to acquire before they can contribute anything useful to the development effort (Brooks, 1975). Why the literature has overlooked such an effect could possibly be attributed explained by Raymond's (2001) suggestion

that Brooks' law, which forwarded the idea of the ramp-up effect, has become obsolete in the context of FLOSS development. The rationale behind this conclusion is the distributed nature of development in FLOSS communities in which the source code itself is considered a medium for communication, thus resulting in the need for fewer interactions between developers. However, should the ramp-up effect really exist, the failure of FLOSS communities to take it into account could have a detrimental effect on their long term sustainability.

To show why these detrimental effects are probable, consider the role the committers play in a FLOSS community. Committers are considered the main drivers of progress in a FLOSS community because they shoulder the majority of the development effort (Shah, 2006; Crowston and Howison, 2005; Mockus et al., 2002). Committers, however, are not guaranteed to remain with a community indefinitely. Many committers have been found to leave after only four months of service with a FLOSS community, only to be replaced by the promotion of contributors that have proven their dedication to and experience within the community (Shah, 2006; Riehle, 2007). Therefore, FLOSS communities need to encourage new contributors to join development efforts in order to ensure that the ranks of the committers are replenished and that their FLOSS communities' progress continues.

Managing the number of contributors is not an easy task because software code base becomes more complex as it evolves over time (Lehman et al., 1997). As a result, committers and/or contributors are required to maintain a strong familiarity with the code if they wish to continue making changes to it (Lehman et al., 1997). The increased com-

plexity of the code base also results in an increase of the ramp-up effect, making it more difficult for new contributors to familiarize themselves with the intricacies of the code design, thus resulting in a decline in the number of new contributors over time.

We argue in this work that the magnitude of the ramp-up effect that prevents new contributors from making an initial contribution is determined by the characteristics of the development knowledge that contributors must obtain before making an effective contribution. Therefore, the ramp-up effect can be viewed as a knowledge barrier (Attewell, 1992) that contributors must transcend before making a useful contribution. While there is empirical support for complexity's negative impact as a knowledge barrier on the promotion of new committers (Midha, 2008), its effect on new contributors is not understood. The value of this work comes from understanding how these knowledge barriers impact the numbers of new contributors from whom committers are promoted (Riehle, 2007). Furthermore, we provide with this work empirical support for the idea that source code structure could encourage participation (Baldwin and Clark, 2006; MacCormack et al., 2006).

New contributors, in particular, will have more difficulty dealing with the growing knowledge barriers associated with software evolution as compared to seasoned contributors or committers, suggesting that the numbers of new contributors could dwindle throughout a FLOSS community's existence as a result of increased development complexity. For this reason, we take a particular interest in new contributors and try to develop a distinction between contributors and committers. More formally, our research question is "What are the factors that lead to greater numbers of new contributors to a

FLOSS community?" We believe that the answer to this question is important because of the role contributors play in sustaining the development effort of FLOSS communities. In addition, the answer highlights the importance of making a distinction between the types of participants in FLOSS communities when building theories of participation.

To answer our research question, we first introduce our conception of the FLOSS marketplace and explain how knowledge acquisition, assimilation, and use are important activities performed by contributors. Next, we explain the costs associated with participation from the contributor's perspective. Then, we shift our focus to the FLOSS community and utilize Transaction Cost Theory (Williamson, 1975) to understand the community-level factors that increase an individual contributor's contribution costs, such as the modularity of the code base and the level of documentation. We then formalize our theoretical model and suggest the means by which we can empirically test this model. Finally, we discuss the potential implications and limitations of this work.

2 Theoretical Framework

The goal of our work is to explain how FLOSS communities can increase their number of contributors regardless of the contributors' motivations. To that end, we first explain how the interaction between a FLOSS community and its contributors can be viewed as a market transaction in which contributors make a source code contribution to the community. We refer to this market transaction as the contribution transaction, which will be discussed in more detail in Sec. 2.2.3.

Assuming that contributors are rational value-maximizing actors, we argue that reducing the costs associated with contribution transactions should increase the number of users that see code contribution as a rational choice, thereby increasing the number of contributors in the community. Furthermore, we make two additional assumptions that set the boundary conditions for our theory. First, we assume that the users of the FLOSS software have some understanding of computer programming and can write patches; therefore, the learning cost for the programming language is not significant enough to prevent a user from contributing. Secondly, we assume that the code base of the FLOSS project is still in active development and has room to grow and create value for its users. A mature project that no one sees a need to further enhance is highly unlikely to attract any new contributors and will not undergo active development.

2.1 The Markets

Markets coordinate the flow of goods and services between two adjacent steps in a value chain. Markets also facilitate the exchange of goods and services between individuals or firms external to one another where the terms of the exchange are determined by market forces. In this latter situation, buyers scan the market for alternatives before making a decision on which terms to go with (Malone et al., 1987). For example, FLOSS developers would assess the technical superiority of a software package before using it and, subsequently, contributing to it to ensure that the software package will continue to serve their future needs (e.g. Ramm, 2008). Such behavior suggests that developers have preferences and choose which FLOSS communities they contribute to, which serves as a demand

force in the FLOSS marketplace. Therefore, for FLOSS communities to gain contributors, they have to improve their software and technical offerings, creating the competitive and supply forces needed to designate the exchange between developers and FLOSS communities as a marketplace. We will discuss this designation in more detail in Sec. 2.2.1.2.

Markets also allow transacting parties to maintain their rights to self govern (i.e., work for themselves) (Conner and Prahalad, 1996). We view the flow of source code contributions between developers and FLOSS communities to be coordinated under a market-type structure. In this FLOSS marketplace, each actor retains the right to self-govern, and the supply and demand forces determine what and who is involved in an exchange, as we shall make clear in Sec. 2.2.1.2.

The FLOSS marketplace and software marketplace are related in that FLOSS communities are participants in both. The two marketplaces can be conceptualized as two adjacent links in the FLOSS value chain in which source code patches that are accumulated by the FLOSS communities in the FLOSS marketplace are integrated into a coherent software system that has value in the software marketplace (see Figure 2.1). FLOSS communities use the FLOSS value chain as an alternative to the proprietary software value chain, which is used by software companies competing with FLOSS communities in the software market. The proprietary value chain differs from the FLOSS value chain in that software companies have hierarchical-like control over developers through employment contracts (Conner and Prahalad, 1996). FLOSS communities, on the other hand, use the market mechanism of the FLOSS marketplace to solicit source code contri-

butions from contributors to maintain the development of the software that is sold/ exchanged in the software marketplace.

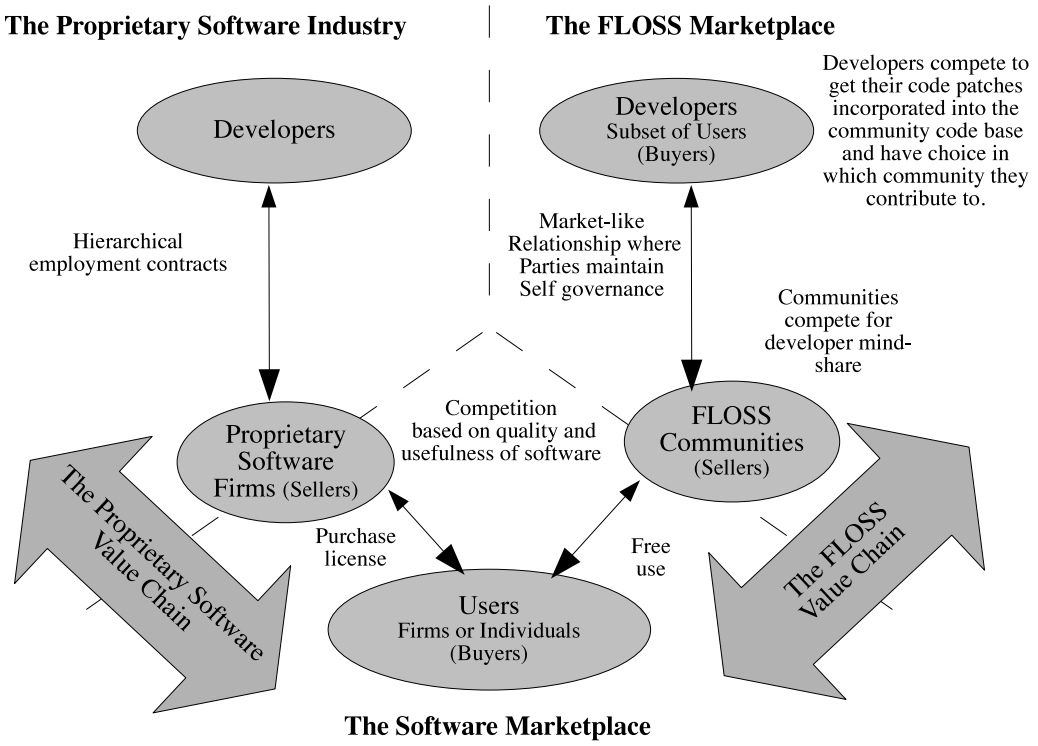

Figure 2.1: The FLOSS value chain

The differences between the software and FLOSS marketplaces, mainly the currency (See Table 2.1), allow us to conceptualize FLOSS communities as sellers in both marketplaces. The FLOSS community is the seller in both markets since they receive currency for their offerings in either marketplace. In the software marketplace, the user[1] (the buyer) engages in a market transaction with a FLOSS community (the seller) when he/ she chooses to download and use its software offering, since the software is made avail-

[1] Developers and users are referred to interchangeably because we are assuming that our theory applies to users that have some understanding of programming to be able to make a contribution.

able for free. In the FLOSS marketplace, however, the developer (the a buyer) engages in a market transaction when he/she offers a patch to obtain the community benefits offered by a FLOSS community (the seller) and that patch is accepted.

Table 2.1: The FLOSS value chain

	Software Marketplace	FLOSS Marketplace
Sellers	FLOSS communities, software vendors, IT solution vendors	FLOSS communities.
Sellers' Goals	Maximize profit, increase market share of product	Maximize patch contributions and quality.
Buyers	Software users, hardware manufacturers or vendors	Individual developers contributing their own free time or organizations that donate the work time of the developers it employs.
Buyers' Goals	Create the highest quality product for the lowest price	Maximize benefit from FLOSS community by having patches accepted beyond satisfying immediate software needs. Benefits include recognition (Lerner and Tirole, 2002; Roberts et al., 2006) and community development assistance (Raymond, 2001; Dahlander and Magnusson, 2005).
Currency	Cash for software	Patches for community benefits.
Market Forces Example	Similar software offerings by different sellers that differ in price and quality.	Different FLOSS communities offering solutions to similar problems that differ in their technical superiority and associated community benefits.

2.1.1 The software marketplace

The software marketplace is considered a marketplace because of the market forces that exist within it (i.e., supply and demand). Demand is generated by users who seek to obtain software that can maximize their returns; that is, users will choose software that meets their feature and quality requirements for the minimum expense. To meet this demand, sellers compete to offer different software choices in order to win as many of users as possible. To remain in business, a seller's goal is to maximize profit from selling software to users by either minimizing the cost of producing the software and thus improve their price offering or by improving their offerings[2] but demanding higher prices. Sellers can afford to remain in this market as long as the income they make covers the expenses associated with developing the software and competing in this marketplace. This model describes the proprietary value chain depicted in Figure 2.1, which is employed by for-profit organizations, such as software development companies. These companies pay the salaries of the developers they contract from the proceeds of the software sales as part of competing in the software marketplace.

FLOSS communities are amongst the competitors for software development companies in the software marketplace. The way FLOSS communities sustain, or fund, their development activities as part of their value chain differs than that of the software development companies. While FLOSS communities compete for a larger user base, just like any other seller in the software marketplace, they do so as a means of attracting contributors to fund their development instead of selling software, which highlights the importance for FLOSS communities to be competitive in the software marketplace. Since any-

[2] Improving a software offering can be done by improving the softwares quality or features.

one can obtain the software produced by FLOSS communities freely without contributing to the development efforts , we argue that the act of contribution follows a different dynamic than the one described in the software marketplace, which relies on the fact that users derive benefits from software usage. This is why we describe the act of contribution in the context of what we refer to as the FLOSS marketplace.

2.1.2 The FLOSS Marketplace

The FLOSS marketplace is considered such due to the existence of supply and demand forces. The sellers in this marketplace are FLOSS communities that supply benefits to contributors associated with community participation, such as peer recognition (Lakhani and Wolf, 2007), development assistance (Dahlander and Magnusson, 2005), and better employment opportunities (Roberts et al., 2006; Lerner and Tirole, 2002). The developers[3] are the buyers in this marketplace, and they will have varying perceptions of value based on the benefits offered by the FLOSS communities. Assuming that developers are value-maximizing actors, they will choose to contribute to communities that offer them the highest returns. The fact that developers have choices regarding which community they can participate with suggests that communities will have to compete with each other by offering greater benefits in order to become the primary choice of the majority of developers. Communities that are able to attract the greatest number of developers—who will eventually become contributors—will have a better chance of sustaining their devel-

[3] We refer to these individuals as developers since they have yet to make a contribution and are so far only potential contributors. See Sec. 2.2 for more detail.

opment efforts. Therefore, the sustainability of a FLOSS community's development effort will be closely associated with its ability to compete in the FLOSS marketplace.

There are numerous behaviors that can be observed in FLOSS communities that suggest the existence of competition in the FLOSS marketplace. We observed that there are numerous software packages that provide solutions for the exact same problem. For example, Ruby on Rails, Django, Zope, Pylons, and Grails are all FLOSS web frameworks that can be used to build web applications. Web frameworks are by no means the only example of competing implementations; the trend continues with almost all software categories, including databases, operating systems, and ERP systems, to name a few[4]. Based on this observation, we conclude that FLOSS communities are not purely cooperative in nature or else we would have observed only a single solution for each problem. Furthermore, we observed that communities actively promote their software. As part of this active promotion, members frequently compare their own community's product to those of other communities (e.g. Ramm, 2008) or even list comparisons on the community websites[5]. Finally, we observed that FLOSS communities often imitate software features and development processes employed by other FLOSS communities that users and potential contributors might have expressed an interest in. These imitations include, for

[4] Both http://wikipedia.com and http://ohloh.net are good sources to identify competing FLOSS implementation for the software categories we mentioned in addition to other categories. Of the identified software categories, we were able to identify at least four competing and successful implementations for each category.

[5] See http://south.aeracode.org/wiki/Alternatives and http://www.cherrypy.org/wiki/CherryPyAndPaste as examples.

instance, the compliance with the WSGI[6] protocol by most Python-based web frameworks, which is perceived to make both development and deployment of web applications easier. The recent move by Wine, Mozilla, and OpenJDK to distributed revision control systems (e.g., Git and Mercurial) instead of centralized systems (e.g., CVS and Subversion) serves as another example of imitation amongst FLOSS communities that serve to improve the development process and appease contributors. We view these actions by communities as a means to gain a competitive advantage or competitive parity in both the software and FLOSS marketplace,s which can be seen as evidence supporting the existence of competitive forces in these marketplaces (Mata et al., 1995).

These observations of the competitive behavior in FLOSS communities might not be sufficient to demonstrate the competition in this marketplace. Therefore, to argue for the existence of this competition more effectively, we need to observe that contributors, as buyers, are willing to change their preferences should FLOSS communities improve their offerings. Furthermore, we need to observe that communities do indeed benefit from improving their competitive position by offering greater benefits to developers, which might include improved product offerings. As it happens, both observations can be made in the FLOSS marketplace. Empirical evidence shows that users and developers show a preference and willingness to migrate between FLOSS projects (Oh and Jeon, 2007), which means that FLOSS communities are able to influence the decisions of users and developers to be part of their community.

---

[6] WSGI stand for web server gateway interface, which is a specification on how web servers and application server communicate.

Furthermore, communities stand to gain much in terms of contributions should a high-profile developer choose to use the software and make significant contributions to improve its appeal to other users and developers. In such a case, it is likely that more contributors would join the development effort of the community. For example, Guido van Rossum, the inventor of the Python language, mentioned in an interview that the Django web framework is the most Pythonic amongst the Python web frameworks (Laporte and DiBona, 2006). Django was incorporated as part of the Google app engine project that van Rossum was working on, which eventually led to his development contributions to the Django project. Ever since, the number of contributors, and even users for that matter, continued to increase for the Django project[7]. Based on these observations in FLOSS communities, we conclude that it is safe to assume that competitive behavior exists in the FLOSS marketplace and that communities stand to gain from being more competitive.

2.2 Sustainability of a FLOSS Community

Based on the FLOSS marketplace framing (See Sec. 2.2.1), we can assume that the competitive forces will drive FLOSS communities to improve their offerings to gain more contributions. The offerings can be in the form of a better solution to a technical problem or benefits associated with a community model of software development. The contributions received by the community would result in the improvement of the software system and would start a self re-enforcing cycle that results in the FLOSS community being more competitive, thereby attracting more contributions.

---

[7] See http://www.ohloh.net/p/compare?metric=Contributors&project_0=Django

While contributions could come from a number of sources when a FLOSS community is highly competitive in the FLOSS marketplace, we are specifically interested in the community's ability to obtain contributions from new sources (i.e., new contributors). It is common knowledge that the development efforts in FLOSS communities is mostly shouldered by relatively few individuals who repeatedly contribute source code patches; these individuals are also known as the core developers and include both committers and frequent contributors (Krishnamurthy, 2002; Mockus et al., 2002; Crowston and Howison, 2005). These core developers do not usually contribute to a FLOSS community indefinitely, and there generally comes a time when they stop contributing altogether (Shah, 2006), assuming of course that there is room for the project to grow and that the decline in participation is not due to maturity[8].

Therefore, to ensure that the development effort is sustained within a FLOSS community, the community needs to take actions to promote the participation of new contributors, thus replenishing the ranks of the lost core developers (Riehle, 2007). Should new contributors cease to join a FLOSS community, there will come a time when the ranks of the core developers will dwindle, thus impacting the community's ability to progress with development. Therefore, we define the sustainability of a FLOSS community as the community's propensity to maintain its development effort over time by effectively converting users into developers.

2.2.1 User Conversion Strategies

[8] We attempt to control for this effect in our sample selection as we explain in Section 2.3.1 and find an insignificant effect for age suggesting that we have appropriately selected our sample.

Assuming that the users are rational value-maximizing actors, FLOSS communities could increase the number of users who provide contributions by making it more rewarding for them to do so. When a good, such as a FLOSS product, is made available for public consumption (Weimer and Vining, 2004), a user has a choice between using the software without incurring any cost related to development or using the software while contributing to its development. Both options allow the actor to benefit from the software, but the rational value-maximizing approach would be to use the software without incurring any contribution costs, which is known as free-riding. Since FLOSS users often exhibit free-riding behavior (Bonaccorsi and Rossi, 2005; O'Mahony, 2003; von Hippel and von Krogh, 2003), we believe it is reasonable to assume that users, in addition to developers who are a subset thereof, are value-maximizing rational actors.

Of the strategies that FLOSS communities can adopt to increase returns for developers, we focus on cost reduction strategies because they have the potential to increase returns for potential participants regardless of their motivations. Attempts to increase the value of the software might improve a FLOSS community's chance of success in the software marketplace, thereby increasing its user base. However, having a large user base does not guarantee an increase in contributors. As rational value-maximizing actors, users need to perceive the value of contributing to be higher than free-riding. As such, reducing the cost of contribution would increase the returns from contribution relative to free-riding, thereby making the option to participate more attractive to a greater number of users. Assuming that the returns developers get from participation vary as a result of both varying cost and value perceptions, we expect the number of contributors in a FLOSS

community to increase as the cost of contribution is reduced. The idea behind this expectation is that reducing contribution costs will increase the returns from contributing for all of the developers relative to other rational options.

Arguing that a cost reduction strategy is effective in attracting new contributors does not imply that benefits of value increasing strategies are ineffective. Indeed, literature focusing on FLOSS participation and motivation suggests that individuals have motivations that differ in nature (e.g., Raymond, 2001;Lerner and Tirole, 2002; Roberts et al., 2006; Lakhani and Wolf, 2007). While value increasing strategies could be effective at targeting specific groups of potential contributors (cf. Lakhani and Wolf, 2007), we are simply making the argument that cost reduction strategies are more effective in that they impact all potential contributor groups regardless of their motivation. The way in which value perceptions are distributed in a community will have an impact however on how many contributors join the development effort when the cost is reduced. For example, due to the nature of different FLOSS communities, there might be a greater number of potential contributors who are close to the borderline of contributing than another community. As such, when both communities reduce the cost to contribute for a similar amount, the number of new contributors might differ.

Given this relationship between value and cost in rational value maximizing decision making and the differing nature of value (i.e., motivation), we find our selves in need of making a simplifying assumption of the nature of the variability in value between different FLOSS communities. We make the assumption that this variability is relatively narrow in range. A result of this assumption is that, as we shall explain, the antecedents to

cost will have a uniform, or relatively similar, impact on participation across communities. Violating this assumption will result in our inability to detect an independent effect for cost, as cost reduction will result in few or no added contributors for communities with a large variability in value perceptions. We will revisit this assumption when discussing the methods used to test our proposed model.

2.2.2 Cost Reduction Strategy

Reducing the cost of contribution involves reducing the cost of development tasks. Software development tasks are considered to be highly complex and knowledge intensive. This means that in order for developers to make initial contributions, they first need to overcome the knowledge barriers of the development process (Fichman and Kemerer, 1997). These knowledge barriers are a significant source of cost that exist because the requisite knowledge required for development is immobile in nature and, therefore, costly to transfer (Attewell, 1992; Choudhury and Sampler, 1997). These barriers are found to be closely related to the complexity of the software and grow as the software matures; as a result, developers will find it easier to join a software development project earlier rather than later in its life cycle (Lehman et al., 1997). Within the FLOSS context, these knowledge barriers include understanding how the source code is organized, how it is compiled, and how modifications can be made to it. As these knowledge barriers increase over time, new developers will find it increasingly difficult to perform these tasks, and as a result, the project will lose future replacements for current team members. Therefore, managing

development knowledge barriers and keeping them low will be very important to ensure the continued supply of new contributors and the sustainability of the FLOSS project.

Since contributors in the FLOSS marketplace benefit in their own way from the exchange of source code for community benefits (Raymond, 2001; Lerner and Tirole, 2002; Shah, 2006; Krogh and Hippel, 2006; Lakhani and Wolf, 2007) and FLOSS communities also benefit from the contributed code, we conclude that this exchange is an economic transaction that can be viewed in light of transaction cost theory (TCT). We refer to this transaction as the contribution transaction and to the knowledge barriers as the main source of contribution costs, which are viewed as transaction costs[9]. A community's ability to manage these transaction costs will increase the number of developers who prefer the contribution arrangement and will, therefore, increase the number of new contributors to the community. We leverage the insights from TCT to gain a better understanding of the main sources of transaction costs and how these costs affect the number of contributors in a FLOSS community. To accomplish this goal, we first need to explain the steps that comprise a contribution transaction.

2.3 The Contribution Transaction

To understand the costs associated with contributions, we first need to review the steps that a developer goes through to complete a contribution transaction (See Figure 2.2) (Raymond, 2001; Shah, 2006; Mockus et al., 2002; Lee and Cole, 2003). A contribution transaction is completed in the FLOSS marketplace when a developer contributes a

[9] From this point on, we will refer to contribution costs and transaction costs interchangeably.

source code patch to a community and the community accepts it as part of its code base. Contribution is a process that occurs over time and includes costs that are predominantly associated with obtaining the requisite knowledge to make a contribution. Since we already assume that the returns developers perceive vary, our theory takes into account the difference in developers' programming abilities, which might affect developers' contribution costs and, therefore, their perceived net returns. Throughout the contribution process, the developer will incur different types of costs when going through the different contribution steps. We group the steps that comprise a contribution transaction into two stages based on the major cost components (See Figure 2.2 and Table 2.2): learning and coordination. Also, it should be noted that these costs are mostly knowledge based (Kogut and Zander, 1996).

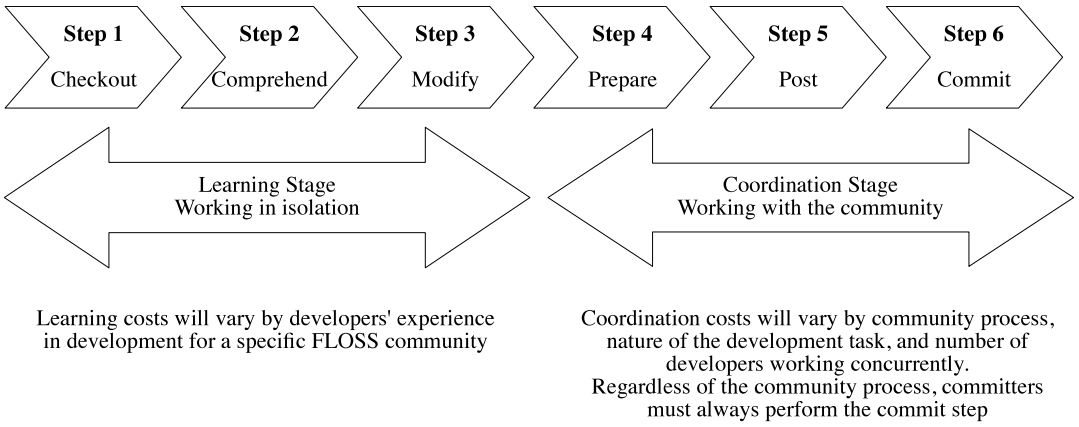

Figure 2.2: The contribution transaction

Table 2.2: Contribution costs incurred by FLOSS developers

	Learning Stage	Coordination Stage
Description	This is the stage in which a useful modification is made to the author's private copy of the community code base. The author could benefit from this modification even though it is not yet integrated into the community code base.	This stage involves integrating the modification made in the initial stage into the community code base. Completion of this stage signals the completion of the contribution transaction and community benefits bestowed on the contributor.
Steps in Figure 2.2	1,2, & 3	4, 5, & 6
Required by	Anyone making a private or public change to the code base	Anyone contributing a patch to the community code base
Recurrence	Required to obtain requisite development knowledge before a contribution can be made to a FLOSS project that is considered new to the developer making the contribution. Incremental effort is expended afterwards for the developer to understand changes made by others.	With every contribution to resolve any conflict.

During the learning stage of contribution, the contributor works in isolation from the community to obtain a modified version of the code base and to implement enhancements that satisfy his/her own needs. Of course, communication could occur between different members of the core development team within the FLOSS community; however, we are interested in the behavior of potential contributors who have yet to be assimilated into the community. As a result, it is safe to assume that potential contributors have yet to under-

stand the community norms or know to whom exactly they need to speak; thus, they are likely to work in isolation. Nevertheless, communicating with other community members during this stage might serve to lower the learning costs. The end product of this initial stage is a usable, enhanced version of the software (Shah, 2006; Ven and Mannaert, 2008) that might create technical conflict if merged with the community code base. However, since the developer of these changes is able to benefit from the software, there is no requirement for him/her to coordinate with the community. For this reason, we assume that the cost to coordinate with other developers is not relevant in this stage.

Even a proficient programmer will be faced with difficulty during the learning phase of contribution. While such individuals may have general programming knowledge, developing knowledge about a specific code base can be very challenging and time consuming (Brooks, 1975; Lehman et al., 1997). Before developers can modify a code base, they need to acquire and assimilate knowledge that is specific to the FLOSS community's code base, which is generally highly immobile in nature (cf. Cohen and Levinthal, 1990; Attewell, 1992; Choudhury and Sampler, 1997). An example of such knowledge would be the detailed programming techniques used throughout a specific code base (Fichman and Kemerer, 1997). Understanding the unique data structures and assumptions about a community's access conditions is another form of source-code-specific knowledge that is important, yet difficult, to acquire. If developers take such knowledge lightly, the software is put in an undesirable state, especially if uninformed changes are made to the code base (Goetz et al., 2006). In addition, how the source code is organized into different files and how these files are brought together to work as a coherent system (Baldwin and

Clark, 2006) is also very important requisite knowledge needed by any developer who wishes to make changes to a code base. Developers incur cost by performing tasks necessary to acquire this requisite knowledge, which involves reading the source code and documentation or communicating with knowledgeable individuals.

The tasks necessary to gain knowledge in the learning stage can be viewed as knowledge barriers that any developer needs to overcome when attempting to make a first contribution to a FLOSS project. Once the requisite knowledge is obtained, the developer can reuse it for subsequent contributions to the same project. Such knowledge cannot be considered non-recurring, since a developer might be required to reacquire such knowledge if enough time is spent away from the project. Given enough time, the general layout of the source code and all of the assumptions embedded in the code will change, making it difficult for the developer to contribute (Lehman et al., 1997).

The costs of the coordination stage differ from the earlier stage because the contributor is no longer working in isolation but interacts with other contributors and committers. The costs associated with the later steps involve coordinating the efforts of contributors working concurrently on the community code base (Malone and Crowston, 1994; Crowston, 1997). Because committers need to approve and incorporate patches into the code base, their capacity to process information might limit the number of patches accepted by a community and, as a result, limit the number of contributors that complete a contribution transaction. For this study, we are interested in the costs incurred by contributors rather than the cost incurred by the community through the committers. As a result, we

will not focus on the committers in our theorizing or control for their effects in our methods[10].

When working simultaneously on the same code base, developers might introduce or remove source code sections that carry assumptions with them that are poorly understood by other contributors or violate patches offered by new contributors (Collins-Sussman et al., 2004; Goetz et al., 2006). For example, consider if developer A contributes a patch that adds a code section that depends on the value of an earlier variable. Developer A makes the assumption that this value will remain constant at the time of execution. Developer B contributes a patch that modifies the value of the variable on which developer A's code depends, thereby violating the assumption developer A made and putting the software in an unstable state that might prove costly to fix. Another possibility is that developer B's patch would remove the variable on which developer A's code depends, thereby breaking the software altogether (See Table 2.3 for example). We refer to such problems introduced by mis-coordination as *technical conflict*, or simply *conflict* for short.

Table 2.3: Example of a technical conflict between developers A and B

Original Code	Developer A	Developer B

`int x = getSomeValue()`
`if(x == CERTAIN_VALUE){`
`  /* do some work on x */`
`  //new value of x`
`  return x;`
`}`
	`int x = getSomeValue()`
	`if(x == CERTAIN_VALUE){`
	`  /* do some work on x */`
	
	`  StoreValueInDatabase(x);`
	`  //new value of x`
	`  return x;`
	`}`
		`int radius = getSomeValue()`
		`if(radius == CERTAIN_VALUE){`
		`  /* do some work on radius`
		`*/`
		`  //Which variable name`
		`  //do we use?`
		`  return newValue;`
		`}`

[10] In study 2 of this dissertation, we take an information processing perspective to examine the relationship between the decision structure of the FLOSS community as represented by the committers, the characteristics of the development task, and the development performance of a community.

Since technical conflict results from mis-coordination, resolving it is a matter of ensuring that both developers share the same knowledge (i.e., assumptions) about their contributions. This involves initiating communication between the developers (Collins-Sussman et al., 2004; Malone and Crowston, 1994; Crowston, 1997) or reading the source code and documentation incorporated within the contributed patch. As with the initial stage, the costs in the later stage are also based on knowledge transfer and acquisition. Therefore, what stands between any developer and his/her participation in a FLOSS community are the coordination costs required to overcome knowledge barriers (Attewell, 1992; Conner and Prahalad, 1996). As such, the main costs associated with contribution are those incurred by the developer to acquire the requisite source code knowledge and the cost of coordination with others to reconcile the technical conflicts that occur when making a contribution to a FLOSS community.

TCT (Coase, 1937; Williamson, 1975) enhances our understanding of how knowledge-related costs influence a FLOSS community's ability to attract contributors. TCT posits that rational transacting parties will choose the most efficient exchange arrangement (i.e., market vs. hierarchy) to complete their transactions. Inefficiency is born out of transaction costs that increase the overall cost of the transaction. From a FLOSS developer's perspective, the choice to participate in the FLOSS marketplace by means of a contribution transaction signals that this is the most efficient choice for that individual. That is, the cost to contribute a patch, including acquiring the requisite knowledge to do so, offers a higher return for that particular developer than any other available options.

According to TCT, transaction costs are necessary for conducting an economic exchange but have no direct bearing on the production costs of the goods or services exchanged (Williamson, 1975). For a software developer, the production costs include the planning necessary to develop a solution and the time spent writing the code. These costs are associated with the direct effort involved in the software production that will result in a usable product. In software development, such costs are usually considered sunk since there is no marginal cost to reproduce another copy of the product (Baldwin and Clark, 2006). Transaction costs are considered to be any additional costs beyond the production costs that are required to sell a software product to a customer (Williamson, 1975). In the case of the FLOSS marketplace, these would include the additional costs for preparing a patch for a specific FLOSS community (i.e., learning and coordination), which would be different if the same software was developed in a different context (e.g., within a software company or a different FLOSS community).

For example, if a software developer creates a database system, all of the effort that went into the creation of the standalone database product is considered a production cost. Any additional effort to adapt this product to embed it into a customer product or to configure it for a customer are additional costs required in order to sell the product to a particular consumer. Since such configuration costs are incurred every time the product is sold to a new customer, they must be incurred before completing each of the sale transactions to the customers, hence the term *transaction costs*. Similarly, in the FLOSS marketplace context, a software developer can develop a software feature as a standalone product, which would be the production cost of that feature. What is key here is that no spe-

cific adaptations are made to accommodate the requirements of a specific FLOSS project. If the developer wants to incorporate his/her feature into a specific FLOSS community code base, then the developer is first required to obtain the knowledge specific to the existing FLOSS community code base (i.e., learning cost) before the developed feature can integrated with that specific code base. This knowledge is considered above and beyond the requirements of production and is, therefore, considered a transaction cost. If the developer wanted to incorporate the same feature into a different FLOSS community, then he/she is again required to learn the knowledge specific to the new community's code base and once more create a specific patch for it. Therefore, transaction costs are born from the effort expended by the developer during the learning stage of the contribution transaction.

In addition to acquiring the requisite code-base-specific knowledge, a developer can incur additional transaction costs from the need to coordinate efforts with other developers working simultaneously on the same code base. When a developer starts working on modifying a code base, other developers would have already committed numerous changes to it. These changes might affect the work of the developer and lead to a technical conflict, but the only way to know if and how a conflict can be resolved requires the developer to acquire knowledge that is specific to the changes made by other developers. There are numerous ways to coordinate the efforts of these developers. For example, they could communicate during development and exchange knowledge that would allow them to create compatible patches and avoid conflicts. However, since no one can be certain who is working on the code base at any one time, the most practical way to coordinate

efforts in FLOSS communities is to rely on a revision control system. A revision control system identifies the most obvious conflict related to an individual overwriting the work of others, and based on the system's findings, the developers can communicate to consolidate the differences in their work (Collins-Sussman et al., 2004). Other conflicts might not be as easy to discover and may manifest themselves as unpredictable behaviors in the software system (i.e., bugs). The effort expended by the FLOSS developers to read through others' work and to communicate with one another to resolve technical conflicts is another source of transaction costs that also involves the transfer of immobile knowledge between different developers.

After showing that the learning and coordination costs can be viewed as transaction costs related to contribution transactions, we can use TCT to understand the sources of these costs. According to TCT, transaction costs exist for two reasons: bounded rationality and opportunism. Of these, we argue that only bounded rationality will be relevant in the FLOSS marketplace context. Bounded rationality implies that the participants in the marketplace have a limited ability to process information (Simon, 1955) and that transaction costs arise from the efforts made by the participants in seeking, acquiring, and processing information that is necessary for the completion of a transaction (Williamson, 1975). In the case of the FLOSS marketplace, transaction costs arise from the cost incurred by the developer to seek, acquire, assimilate, and use the knowledge that is unique to a FLOSS community's code base, which involves information processing and communication (Turner and Makhija, 2006; Choudhury and Sampler, 1997). Therefore, developers will be limited in their ability to read source code or documentation, to seek and so-

cialize with knowledgeable individuals, and to coordinate development efforts with other developers. Developers who lack sufficient time or the capacity to incur the cost of such activities will not be able to participate in a FLOSS project. Therefore, a FLOSS community could reduce transaction costs if actions are taken to reduce the need for, or the cost associated with, the aforementioned information-processing activities performed by developers.

According to TCT, the other reason transaction costs exist is due to opportunism. Opportunism implies that there will be participants in the marketplace who will exploit any opportunity that is presented to them to gain higher returns, even if doing so is against the interest of others. This suggests that some market participants will avoid fulfilling contract obligations if they can do so without facing negative consequences. Transaction costs are incurred when either party involved in the transaction expend effort to monitor and prevent opportunistic behavior by the other party. For opportunism to be relevant to our current study, it must impact the transaction costs incurred by the contributors.

2.3.1 Opportunism in FLOSS

Academic literature on FLOSS identified two forms of opportunistic behavior: free-riding and commercial appropriation (von Hippel and von Krogh, 2003; O'Mahony, 2003). Free-riding is the act of using the software without contributing to its development, while commercial appropriation is the act of profiting from the software without compensating the developers for their effort. With regards to free-riding, developers choose to participate in a FLOSS community knowing beforehand what the FLOSS license entails;

namely, that FLOSS-licensed software does not place any restriction on usage, even if users choose to free-ride. Furthermore, free-riding behavior does not have any significant negative impact on either the community or its contributors because the software is a digital good and its value does not decrease nor do its costs increase with additional usage (von Hippel and von Krogh, 2003).

As for commercial appropriation, the General Public License (GPL) was drafted specifically to prevent such behavior (O'Mahony, 2003). When an entity appropriates the effort of developers, a license violation occurs. Such violations, when addressed, are usually dealt with by the non-profit organization that oversees the interest of the community (O'Mahony, 2003). Hence, individual developers do not shoulder any of the burdens that could potentially deter them from contributing. In other words, we do not expect the cost associated with monitoring and preventing opportunistic behavior to have any significant impact on the decision or ability of FLOSS users to participate in the development effort.

Appropriation might be a bigger issue when deciding to release a software package under a FLOSS license because a for-profit organization stands to lose all of its future income from directly selling the software (West, 2007). However, appropriation might be less of a concern for any organization considering contributing to an ongoing FLOSS effort by providing bug fixes and incremental feature enhancements. The cost of such activities would be a negligible because the total development effort would be distributed amongst the community members. More importantly, the Open Source movement, as represented by the Open Source Initiative, approves of licenses that permit appropriation. Permissive licenses, as they have come to be known, entice commercial participation be-

cause they tie the success of the commercial entity with the success of the FLOSS community (AlMarzouq et al., 2005). Empirical evidence suggests that FLOSS communities that use permissive licenses (i.e., licenses other than the GPL) have increased contributions and sponsorship from for-profit organizations (Stewart et al., 2006).

We are not trying to argue that opportunism does not exist in the FLOSS context; on the contrary, the need for FLOSS licenses and the existence of non-profit organizations to look after the best interests of FLOSS communities suggests that it exists (von Hippel and von Krogh, 2003; O'Mahony, 2003). What we are arguing, however, is that opportunism will not be a significant force on a user's decision to make a source code contribution to an already established and active FLOSS project. In support of our argument, consider the main factor that gives rise to opportunism-based transaction costs, namely asset specificity (Williamson, 1975; Clemons and Hitt, 2004). When a transacting party makes an investment that is valuable only as part of a specific transaction, the other transacting party is the only entity that would be able to take advantage of the situation. As a result, transaction costs arise from the need to monitor and prevent transacting parties from taking advantage of one another.

In the FLOSS marketplace context, when a developer makes a contribution, the community has no way of forcibly (i.e. control) extracting any benefits from a contributor beyond what the contributor offers willingly. More importantly, when the software is licensed as FLOSS, a contributor is guaranteed to benefit from the software in the future; therefore, no entity can hold any contributor hostage to his work by threatening to prevent him/her from benefiting from the software. Should such a situation occur, the con-

tributor could simply fork[11] the source code or switch to another project. Finally, the parties involved in the contribution transaction (i.e., the FLOSS community and the contributor) have congruent goals, which mitigates the need to monitor opportunistic behavior (Ouchi, 1980; Clemons et al., 1993). The goals of the community and the contributor are congruent because the community relies on contributions, and the contributor, as a user, would like to see the software improved and maintained. This symbiotic relationship enables both parties to benefit from the act of contribution. Therefore, we conclude that opportunism-based transaction costs are not significant in the FLOSS marketplace context.

Although opportunism is not relevant to understanding users' contributions to FLOSS, it does not mean that TCT is inappropriate for understanding contributions in the FLOSS marketplace, as transaction costs can and do exist in the absence of opportunism (Conner and Prahalad, 1996). According to Clemons et al. (1993), coordination costs and transaction risks are the two main dimensions of transaction costs. Coordination costs are costs born out of the bounded rationality condition, and transaction risks are costs associated with mitigating the opportunistic behavior of the other party in an exchange. Therefore, transaction costs in the FLOSS marketplace are purely coordination costs, which Clemons et al. (1993) define in the most liberal form to encompass all costs born out of bounded rationality. In our case, these costs include those related to acquiring the requi-

[11] Forking occurs when a developer makes a copy of a source code and starts a new development branch. Every developer forks the community's source code before developing a feature that is eventually contributed to the community. Forking can also occur at the community level when members disagree on fundamental issues. The result is that the community is split with each half working on a different copy of the original code base.

site development knowledge and coordination with other developers. This idea is consistent with our view that the costs associated with a contribution transaction are knowledge based, requiring information-processing efforts on behalf of the developers. We will clarify the details of TCT further in the subsequent section as we identify the causes of transaction costs in the FLOSS context. Thus far, we have explained how contribution transaction costs are mostly knowledge and coordination based (Kogut and Zander, 1996). As a result, it will be important to understand the characteristics of knowledge that are related to the effort expended by individuals to learn and coordinate (Grant, 1996b).

2.4 Community-Level Antecedents to Cost

After establishing that transaction costs in the FLOSS marketplace are knowledge based and born out of bounded rationality, it becomes important to understand the characteristics of the requisite development knowledge and how it might impact the cost of contribution. Doing so allows us to understand how a community could manage source-code-related knowledge in order to minimize the transaction costs incurred by its developers. We follow the classification proposed by Turner and Makhija (2006), which is based on an extensive review of knowledge management literature. This classification states that the main observable dimensions of knowledge are codifiability, completeness, and diversity. We also work under the assumption that the source code that resides in a FLOSS community is an explicit form of the knowledge required for development and that the characteristics of this knowledge can be observed from the source code. In the following section, we will use TCT to explain how the dimensions of knowledge relate to transac-

tion costs, and we will identify the salient community-level characteristics that could impact developer-level transaction costs (See Figure 2.3 and Table 2.4 for an overview).

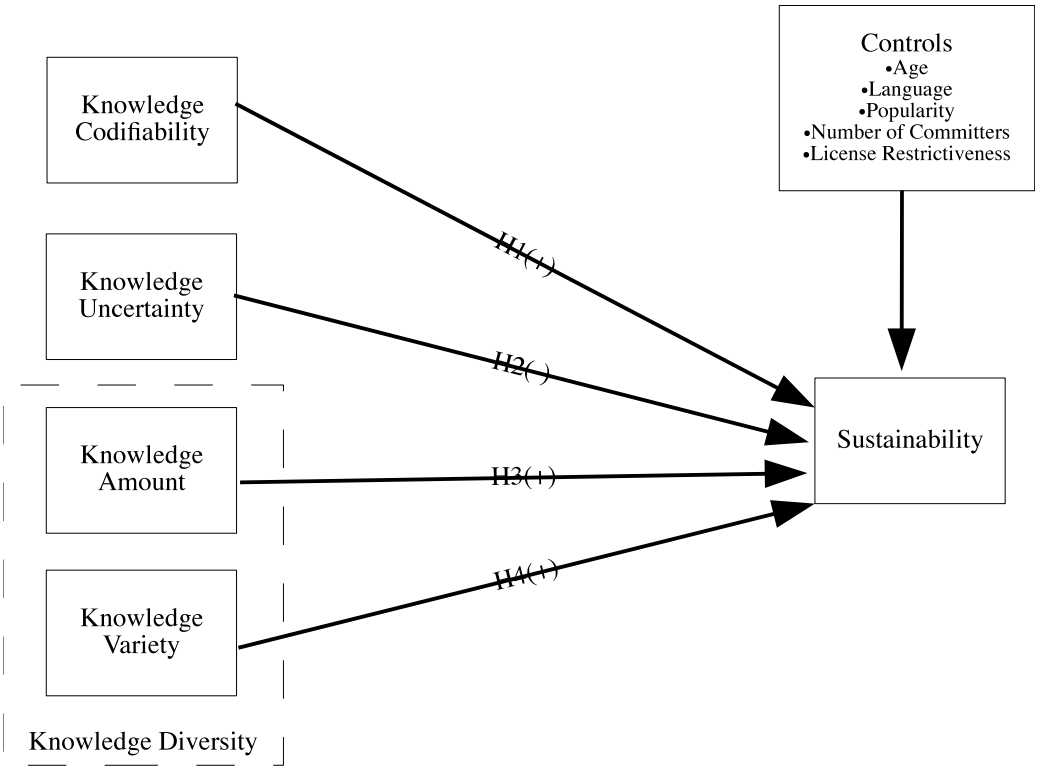

Figure 2.3: Overview of research model

Table 2.4: Overview of theoretical constructs

Construct	Definition
Sustainability of FLOSS Community	The ability of the FLOSS community to continue to maintain the development effort over time by effectively converting users to developers.
Knowledge Codifiability	The extent to which the development-related source code knowledge is articulated, unambiguous, and observable.
Knowledge Completeness	The degree to which the available knowledge for the software development task is entirely sufficient to predict the behavior of the software system after modification.
Knowledge Diversity: Amount	The number of distinct knowledge domains from which the requisite development knowledge to contribute to a FLOSS community draws.
Knowledge Diversity: Relatedness	The extent to which the different knowledge domains from which the requisite development knowledge of a FLOSS community draws are related.

2.4.1 Sustainability

Sustainability, the endogenous variable of this study, is defined as a FLOSS community's ability to continue to maintain the development effort over time by effectively converting users to developers. As explained in Sec. 2.2.2, we expect that FLOSS communities with low contribution transaction costs will observe greater numbers of new contributors as compared to communities with high transaction costs. We attribute this tendency to a greater proportion of potential contributors that see contribution as a rational choice relative to free-riding. Since the complexity of software and contribution transaction costs are expected to increase over time (Lehman et al., 1997), we argue that communities are able to increase the number of new contributors over time if they are able to maintain low, or

lower, contribution transaction costs. We summarize how the main constructs impact the components of contribution costs (i.e., learning and coordination costs) in Table 2.5.

Table 2.5: Antecedents to coordination costs

Construct	Learning Cost	Coordination Cost
Knowledge Codifiability	Reduce effort in comprehending the source code by making explicit knowledge that is important to comprehend the source code.	
Knowledge Completeness	Incomplete knowledge (i.e., uncertain) is a result of the dynamic nature of the code base and will require contributors to expend additional effort at time of committal to comprehend changes made since that last source code checkout was performed.	When committing to a dynamic code base, a contributor might break the work of others that has already been committed and will require coordination with these community members to resolve any incompatibilities with the contributed patch.
Knowledge Diversity: Amount	To work on an isolated programming unit of the code base (i.e., module), a contributor needs to at least understand how this module interacts with other modules in the code base. The effort to acquire such knowledge grows with the number of modules in the code base.	
Knowledge Diversity: Relatedness	Comprehending larger modules that result from highly related knowledge will require greater effort from contributor.	Working on larger modules that result from highly related knowledge is likely to result in the need of one or more developers to work on the same modules and result in technical conflict.

2.4.2 Knowledge Codifiability

Knowledge codifiability refers to the extent to which knowledge can be articulated into indisputable, unambiguous, and observable information (Kogut and Zander, 1992).

Highly codifiable knowledge is referred to as explicit knowledge (Zander and Kogut, 1995; Turner and Makhija, 2006) and can be transferred easily between individuals without losing meaning (Grant, 1996b, 1996a). Non-codifiable knowledge, referred to as tacit knowledge (Turner and Makhija, 2006), is more difficult to transfer between individuals, as it is difficult to articulate and requires extensive communication amongst individuals before it can be transferred between them (Alavi and Leidner, 2001).

An important aspect of any FLOSS code base is the amount of documentation that resides within it. These comments are a codified form of the tacit knowledge that resides in the minds of the developers. Such knowledge could include assumptions about variable access rules that source code might not clearly convey. Since such knowledge is only useful for a specific project, it is considered asset specific, meaning that it is of no value outside that specific project (Williamson, 1975).

Maintaining source-code-related knowledge in a tacit form (i.e., in the minds of developers) would make the transfer of such knowledge expensive (Zander and Kogut, 1995; Grant, 1996a). This type of specificity is referred to as human-asset specificity because the knowledge is specific to the individual who obtains it (Williamson, 1975; Malone et al., 1987). Communicating such knowledge requires significant investments from developers, which could involve time and effort searching for and socializing with other community developers to share and acquire the requisite development knowledge. Since the knowledge obtained is specific to the FLOSS project, most of it will be useless for developers in other projects, which would make it difficult for many developers to justify

expending the effort to obtain this specific knowledge when the effort needed to acquire it is significant.

When a FLOSS community expends the effort and codifies this knowledge by embedding it as comments within the code base, the knowledge becomes easier to communicate. It is no longer specific to certain individuals; rather, any individual has the ability to obtain it simply by reading the documentation. Knowledge in its explicit form as documentation[12] becomes easily searchable and transferrable. Even though this knowledge is still specific to the FLOSS code base, it is now less human specific and much easier to obtain and comprehend by more developers than when it was in a tacit form. This, in turn, reduces the amount of transaction costs incurred by developers wanting to obtain this knowledge. Therefore, we conclude that communities that expend greater efforts in documenting the code base will reduce transaction costs related to contribution, thereby gaining a greater number of new contributors over time. As such,

H 1 *The extent to which source-code-related knowledge for a FLOSS community is codified will be positively related to the sustainability of that community.*

2.4.3 Knowledge Completeness

Knowledge completeness is defined as whether the available knowledge for decision making or task completion is entirely sufficient (Turner and Makhija, 2006). It is considered the mirror image (i.e. opposite) of knowledge uncertainty, which is defined as the unpredictability in the knowledge that is a result of its dynamic nature (Turner and Ma-

[12] We use the terms documentation and comments interchangeably.

khija, 2006). The outcome of any task in which the individual performing the task possesses complete knowledge about it becomes predictable. For example, a manufacturing worker that is required to tighten a specific screw on a product need only know how to use a screwdriver and which screw to work on. This task is simple and requires limited knowledge to perform, which makes it easy for the worker to know everything there is about tightening that specific screw for that specific task. The worker will always know what happens when the screw is tightened, which makes the outcome of the task highly predictable.

On the other hand, when the knowledge available to an individual performing a task is not complete, the outcome of the task becomes unpredictable. Knowledge associated with software development tasks are considered incomplete because developed software usually behaves unpredictably. While the developer might have honestly attempted to develop software to meet certain specifications, the nature of software development introduces many uncertainties that might break the software that the developer may not have accounted for. For example, a developer might produce software that works perfectly in the development environment and on a specific hardware configuration. However, once this software is released commercially, the developer may start receiving complaints from customers that the software does not work on their particular hardware configurations. Although the developer might know how to deal with such problems once they arise, at the time of software development, the developer did not think that the customers' hardware configurations would matter. Thus, the developers' knowledge was incomplete, and the outcome of the task was unpredictable.

In the FLOSS marketplace context, the developers are boundedly rational and have no way of knowing what any other developer is doing until a conflict arises. As described in Sec. 2.2.3, contributors who have not absorbed the norms of the community are not able to communicate or coordinate development activities as effectively with other members. This uncertainty gives rise to technical conflicts (e.g., developers overwrite the work of one another) that need to be resolved before a patch can be fully committed to the code base (Collins-Sussman et al., 2004). Therefore, developers who do not possess the complete knowledge that allows them to finish their development task with certainty will be required to expend additional effort in examining the contribution logs to understand what other developers have done and to communicate with them to resolve any technical conflicts. This leads us to conclude that there is a transaction cost associated with every contribution transaction due to a lack of complete knowledge, which requires a developer to expend extra effort before a patch can be fully committed.

According to TCT, uncertainty is a primary cause of transaction costs (Williamson, 1975). In the absence of opportunism, unforeseen changes in the environment that are born out of uncertainty can lead to honest disagreement between transacting parties who are boundedly rational since each of them would possess incomplete, but different, knowledge. The effort expended from the two parties to acquire more complete knowledge in order to reach an agreement is the main source of transaction cost (Conner and Prahalad, 1996). Similarly, technical conflict in a FLOSS community could occur between two developers when they do not have complete knowledge about what each developer is doing. Therefore, to reduce the transaction costs incurred by developers from

the occurrence of technical conflicts, a FLOSS community should design its code base such that the outcomes of development tasks are more predictable, thereby reducing the chance of technical conflicts.

Designing the code base to be modular is one approach to reducing the uncertainty associated with developing a code base and to reducing technical conflict (Baldwin and Clark, 2000; Tiwana, 2008). A software system is said to exhibit modularity if its parts can be developed independently of one another but still work well together as a whole (Baldwin and Clark, 2006). This approach entails limited dependency between parts (i.e., loose coupling) and a high level of dependency between the components that comprise a single part (i.e., cohesion) (MacCormack et al., 2006).

Making changes to modular code bases is inherently less certain because side effects are usually isolated to the modified module (Baldwin and Clark, 2000; Jackson, 2006). In addition, modular code bases reduce the cognitive burden on developers, since developers will not be required to comprehend the complexities of implementation of modules that are not relevant to their needs (Jackson, 2006; Darcy et al., 2005). Finally, modularity allows different developers to work in parallel on different modules and be able to easily integrate their work together with out the need to know what other developers are working on (Baldwin and Clark, 2000; Sanchez and Mahoney, 1996; Tiwana, 2008). Modular structures bring about these benefits by maximizingwithin modules and dependencies between modules (Darcy et al., 2005; MacCormack et al., 2006).

What is critical to obtain the benefits of modularity is that the structure of the code base (i.e. dependencies) needs to remain consistent. With a stable structure developers

can work on specific modules knowing that if they do not change the way in which the module interacts with other modules, their modifications will continue to work once integrated with the main code base (Darcy et al., 2005). Such stability is said to convey more complete knowledge to developers because the developer can be certain that his/her work will continue to work after changes are completed. Stability in the FLOSS context would mean that by the time a contributor prepares his/her patch, it is likely that no additional effort is needed to integrate the patch to the community code base. As such, the source code is said to convey complete development knowledge when it is checked out by the contributor.

On the other hand, if the code base of a FLOSS community was highly dynamic, then the community code base is likely to mutate significantly in the from the time a copy of the code base is checked out by a contributor to the time a patch is submitted for committal. To integrate such a patch, the contributor is required to exert extra effort to comprehend what changes were made to the community code base and how to modify the patch to work with such changes. Such code base is said to convey highly uncertain knowledge (i.e., incomplete) when it is checked out by the contributor (Turner and Makhija, 2006). Therefore,

H 2 *The extent to which source-code-related knowledge for a FLOSS community is complete will be positively related to the sustainability of that community.*

2.4.4 Knowledge Diversity

Knowledge diversity is defined as the amount and relatedness of the knowledge required to complete a software development task that is equivalent in functionality to what currently resides in the FLOSS community (Turner and Makhija, 2006). According to Turner and Makhija (2006), diverse knowledge overlaps with the notion of complex knowledge as described by Zander and Kogut (1995) but goes further to incorporate the notion of relatedness. Knowledge amount refers to the number of distinct knowledge domains from which knowledge draws, whereas relatedness refers to how difficult it is to decompose knowledge into independent parts. Knowledge that is highly related resides entirely in the mind of a single individual, while the parts of highly unrelated knowledge could be dispersed over the minds of numerous individuals (Galunic and Rodan, 1998). Turner and Makhija (2006) argue that the notions of relatedness and amount are highly interrelated and difficult to separate because knowledge drawing from a larger body of knowledge is also likely to be unrelated. Nevertheless, we believe that each dimension of knowledge diversity (i.e., amount and relatedness) will have its own unique impact on the cost to contribute to a FLOSS community as we shall explain in the next section. Therefore, we will discuss the impact of the dimensions of knowledge diversity on the sustainability of a FLOSS community independently.

Knowledge Amount

We define knowledge amount as the number of distinct knowledge domains from which the requisite development knowledge to contribute to a FLOSS community draws. Development tasks that draw from a broad base of functional domains are referred to as

complex tasks (Zander and Kogut, 1995). Simple and highly specialized tasks will draw from a narrow body of highly related knowledge and will require little effort to complete (Grant, 1996a). Given that a source code is an explicit form the requisite development knowledge, the complexity of the software product will reflect the complexity of the development task.

In the FLOSS marketplace context, the development of complex products will not only require understanding general programming principles, but depending on how general the software product is, developers might also need to draw on knowledge from different disciplines. An Enterprise Resource Planning (ERP) system serves as a good example of a general software system that draws from a broad and complex knowledge base. In addition to understanding how to build robust software systems, developers need to draw on knowledge from other disciplines, such as accounting and resource management, in order to build an ERP system. There is also the possibility of needing to implement knowledge about different programming languages to integrate an ERP with COBOL-based legacy systems. A more specific software system, such as a text editor, will draw from a much narrower and simpler knowledge base. Internalizing the highly diverse knowledge required for the development of complex software products will be impossible for a single developer to do (Grant, 1996a) and will be a formidable task even for a group of developers.

Well designed code bases, as generally found in FLOSS projects (MacCormack et al., 2006), could reduce the cognitive burden on developers and only require them to comprehend a subset of the code base (Baldwin and Clark, 2006) by encapsulating (i.e.

grouping) the code of related functionality into independent parts (i.e. modules) (Page-Jones, 1998; Darcy et al., 2005). However, as we have explained in Section 2.2.4.3, developers still need to obtain a high level understanding of the interplay between the individual parts of the software system and how they might relate to the part they are working on before they can modify it (Darcy et al., 2005). Doing so, however, becomes more difficult as the software continues to grow and draw from an even greater number of knowledge domains (Lehman et al., 1997). The effort required to reach an understanding that would allow a potential contributor to make a modification to the code base will increase as the number of individual parts that comprise the software increase. As a result, potential contributors are required to invest more upfront effort before making their initial contributions, as the requisite development knowledge amount increases. Therefore, we expect knowledge amount to have a negative relationship with the sustainability of a FLOSS community. Hence,

H 3 *The amount of source-code-related knowledge that exists in a FLOSS community is negatively related to the sustainability of that community.*

Knowledge relatedness

While we have discussed in Section 2.2.4.3 numerous benefits that are a direct result of modularity, such as the reduction of development uncertainty (Tiwana, 2008) and enabling parallel development (Baldwin and Clark, 2006), these benefits are mostly a result of the loose coupling of the modules that comprise the code base. In regards to knowledge relatedness, we would like to get at the effort associated with comprehending a sin-

gle cohesive module (Darcy et al., 2005). Highly modular code bases may be equivalent in terms of the number of modules and the degree of coupling between modules; however, they could vary greatly in terms of the size of each individual module (See Appendix A for a clear delineation of the modularity-related constructs in our study.) We attribute this variability in size to the relatedness of the underlying knowledge.

Knowledge relatedness refers to the extent to which the different knowledge domains from which the requisite development knowledge draws are related. When the requisite knowledge is highly related, it must be wholly comprehended in the mind of a single individual (Galunic and Rodan, 1998). However, if the development knowledge is highly unrelated, it can be easily decomposed into independent parts that different individuals are able to comprehend, thereby making it possible for different individuals to work on relevant parts of the source code independently. Assuming that the source code is an explicit form of knowledge and that FLOSS communities are effective in designing the source code to reflect the underlying knowledge, then highly unrelated knowledge will result in modules that encapsulate functionality that draws from a single domain. As relatedness increases, these modules will draw from a greater number of related domains and result in larger modules (Darcy et al., 2005).

The increased module sizes will also have a negative impact on coordination costs as the encapsulation of multiple functionalities will likely create the needs for multiple developers to work on the same module. As a result, a greater need for coordination arises whenever contributors are required to work on the same module (Crowston, 1997). The

increase in coordination effort contributes to the overall increase in contribution costs and thus negatively relates to sustainability.

When there are two code bases equivalent in modularity and amount of knowledge but one code base has larger modules than the other due to higher knowledge relatedness, new contributors are likely to expend less effort contributing to the code base with the smaller modules. Assuming that the contributor needs to comprehend the underlying knowledge of only a single module to make a contribution, the effort needed for the smaller module will be less difficult to acquire than for the larger one. The same could also be said of the need for coordination, as larger modules encapsulated a greater number of functionality will create the need for contributors to work on the same module and result in greater coordination effort. As a result, we expect knowledge relatedness to increase contribution costs because it increases the size of modules in the code base that has an adverse effect on both coordination and learning costs. Hence,

H 4 *The relatedness of source-code-related knowledge that exists in a FLOSS community is negatively related to the sustainability of that community.*

3 Methodology

3.1 Sample

Prior studies on FLOSS chose their sample from the projects listed on sourceforge.net since it was the most accessible data (e.g. Krishnamurthy, 2002; Crowston and Howison, 2005; Stewart et al., 2006; Liu and Iyer, 2007; Midha, 2008). However, such data is not

fit to test our theory given that many projects listed on sourcefore.net are not popular or actively being developed (Krishnamurthy, 2002). Because we assume that the FLOSS projects to which our theory applies have inherent value to the users which could result in contribution, we limit our sample to include successful FLOSS projects that have proven their worth. However, we needed to take care when selecting observations from this sample, as we did not want to include inactive projects, either from lack of interest or maturity, as this might bias our results since such projects will not have new contributors regardless of the variation in our hypothesized effects. In addition, since the act of participation requires the project to be known, projects that are popular are likely to attract more users and contributors. Therefore, we needed to select FLOSS projects that were equally popular, or we needed to be able to assess and control for the popularity of the projects we chose for our sample.

Ohloh.net met the requirements that we placed on our sample. With more than 275,000 listed projects, it offered the best representative sample of the FLOSS population. Unlike sourceforge.net, which provides data for only projects that it hosts, ohloh.net covers projects hosted independently or those hosted by services like sourceforge.net. In addition, Ohloh.net provides a meta-community of developers that self report usage and contribution, which can be used as a means of estimating the popularity of the projects.

To test our hypotheses (see Figure 2.3), we sampled from the top 1000 most popular FLOSS projects listed on ohloh.net that enjoyed active development between the beginning of January 2007 and the end of June 2009. Of these projects, we found only 22% to

be hosted on sourceforge.net, confirming our claim that ohloh.net offers a more representative sample of popular FLOSS projects.

Because it was impractical to analyze all of the projects in the top 1000 and to maximize the external validity of the empirical findings from our sample, we opted to analyze projects that use one of the three most popular programming languages in FLOSS development: C, C++, and Python. Between them, these programming languages were used by over 18% of all of the listed projects on ohloh.net and by 47% of the projects in our sample frame. Furthermore, given the fact that Python is a dynamically typed scripted language and that C and C++ are statically typed compiled languages, we believe that we have a good representation of the most widely used programming paradigms within FLOSS communities.

We excluded from our sample meta-projects that included more than a single project, such as the Gnome and KDE projects. However, we analyzed projects under these communities if they were popular enough to be listed independently in the top 1000 projects on ohloh.net. From our sample frame, we identified 289 potential projects to analyze, which represented 28.9% of the top 1000 projects. Once our sample was identified, we proceeded to download the source code repository for the projects on which we performed our analysis. There were however five projects that were impractical to analyze using our methods given their extremely large code bases and were, therefore, excluded. In addition, a number of projects had incomplete development history or missing values for the variables used in our study, which is why we ended up with a total sample size of

235 projects. Below is a summarization of the steps we took to obtain and analyze our sample:

1. Download list of top 1000 projects listed on ohloh.net to ensure we analyze valuable and popular FLOSS projects.
2. Download source code repositories for C-, C++-, and Python-based discrete FLOSS projects.
3. Extract quarterly data points for variables listed in Table 2.6.
4. Screen the data for missing values and periods of inactivity that could bias our results.
5. Prepare data.
6. Perform statistical analysis.

Table 2.6: Variable operationalizations

Construct	Definition	Operational Definition	Operationalization
Sustainability	The ability of the FLOSS community to effectively convert users to contributors in order to maintain the development effort over time.	The number of new unique contributors to the FLOSS project.	Estimated as the count of new individuals joining the development effort identified from the revision control system during the analysis period without reference to the contributors in the previous periods (Midha, 2008).
Knowledge Codifiability	The extent to which development-related source code knowledge is articulated, unambiguous, and observable.	The extent to which the source code base is documented.	Estimated as the line count of documentation in the source code as a ratio of source lines of code for the beginning of the analysis period (Zander and Kogut, 1995).
Knowledge Completeness	The degree to which the available knowledge for the software development task is entirely sufficient to predict the behavior of the software system after modification.	The magnitude of change in the structure of the code base.	The change in graph modularity measure (Newman and Girvan, 2004; Newman, 2006b) that results from partitioning the source code dependency graph using the leading eigen-vector method (Newman and Girvan, 2004; Newman, 2006a) for the beginning of the analysis period.

Knowledge Diversity: Amount	The number of distinct knowledge domains from which the requisite development knowledge to contribute to a FLOSS community draws.	The number of unique modules in the code base.	Estimated as the count of modules identified by the leading eigen-vector method for the beginning of the analysis period (Newman and Girvan, 2004; Newman, 2006a).
Knowledge Diversity: Relatedness	The extent to which the different knowledge domains from which the requisite development knowledge of a FLOSS community draws are related.	The increase in average module size as a result of having high knowledge relatedness.	Estimated as the count of source code lines for the beginning of the analysis period as a ratio of number of modules (Darcy et al, 2005).

3.2 Variables

Since our theory involves learning processes in which new developers overcome knowledge barriers to become contributors, we expect time to play an important factor (Attewell, 1992). Once our project sample was identified, we obtained quarterly observations for our variables between January 2007 and June 2009. Since some of the projects were started after January -2007, we had an unequal number of observations per project. We excluded from each project the first observation because we used it as a reference for subsequent observations to determine the value of some of the variables. In addition, the last observation from each project was discarded because some of the variables were based on the difference between the value of current and subsequent periods. In total, we had 1832 total observations. Table 2.7 gives some descriptive statistics about the projects

we analyzed. As can be seen, the quarterly observations suggest that the projects in our sample were actively developed and having at least a single patch commit performed per quarter. Since our subsequent analysis also suggests that there is no significant effect for the age of the project, we can safely assume that our results are not impacted by inactivity due to maturity.

Table 2.7: Descriptive statistics for project sample

	Median	Mean	STD
Age In weeks relative to 1-1-2007	4	5.64	48.7
Popularity	57.5	192.1	447.71
Committers Count per quarter	5	10.84	14.489
Contributor-S count per quarter	8	16.29	26.786
Commits count per quarter	116	289.2	456.656

3.2.1 Sustainability

Content Validity

Sustainability is defined as a FLOSS community's ability to effectively convert users to contributors in order to maintain development efforts. Since FLOSS developers do not remain with the community indefinitely, sustainability will center on a community's ability to attract new contributing members. In our sample, new contributors can be identified from the log messages in the revision control system of the FLOSS community. The revision control system keeps track of all of the commits made to the community code base

and associates every commit to a committer. Identifying the contributors is a matter of parsing the log message prepared by a committer for any indication that the patch was contributed from someone else, such as a name, pseudo-name[13], or email. Individuals with no references to them as contributors or committers in prior log messages are considered to be new contributors.

Procedure

Given how complex the procedure was, we provide a high-level summary of the main steps we performed to extract the number of new contributors and offer a more detailed description in Appendix B:

1. Extract contributor names for projects that list names in log messages:
    (a) Perform manual extraction of names for randomly selected sample.
    (b) Perform automated extraction for the same sample as the manually extracted sample.
    (c) Assess Inter Class Correlation (ICC) (Shrout and Fleiss, 1979) between manual and automated name extraction results and optimize automated process for increased reliability (ICC = 0.92).
2. Extract contributor names for projects that list contributor names in project tracker:
    (a) Extract ticket numbers from log messages.

[13] A pseudo-name is a name used to identify an individual that is not his/her real name. Internet users use such names to maintain anonymity or make it more convenient for them or others to type or remember their names.

(b) Parse ticket webpage and extract name of contributor and the contribution date.

3. Clean names by removing any extracted value that does not represent name, pseudo-name, or email.
4. Normalize names such that misspelled names and related names refer to a single individual.
5. For each analysis period, count new normalized names having no reference to them in prior analysis periods as a single new contributor.

3.2.2 Knowledge Codifiability

Content Validity

We defined knowledge codifiability in Sec. 2.2.4.2 as the extent to which the development-related source code knowledge is articulated, unambiguous, and observable. The source code itself is a form of codified knowledge; however, a develop cannot internalize this without obtaining certain requisite knowledge (Grant, 1996a; Fichman and Kemerer, 1997). For example, there might be some assumptions about data structure access or organizational conventions that, if violated, would render the source code unstable. Such information can be embedded as documentation in the source code, which would be considered an explicit form of the requisite knowledge needed for development. Such documentation could also include explanations for some of the source code that developers could reference when they need to.

Procedure

To get an indication of the extent to which requisite knowledge was codified, we counted the lines of code comments that were available for the developers. We used the ratio of lines of comments to lines of source code as an estimate of the extent to which the development-related source code knowledge is codified. We obtained this estimate for the beginning of the analysis period (i.e., the estimate was obtained by analyzing the source code from the commit that occurred closest to and after the first day of the quarter to be analyzed).

3.2.3 Knowledge Completeness

Content Validity

As mentioned in Sec. 2.2.4.3, knowledge completeness refers to whether the available knowledge for the software development task is entirely sufficient to predict the behavior of the software system after modifications. Thus, it seems likely that tasks that are highly uncertain would not have enough available knowledge to complete them. For FLOSS software development, this not only involves the knowledge to develop the software but also the knowledge to determine that the software is fit for use after a change is made (i.e., that the change will not introduce a bug or break the system). When the development task is highly uncertain, the developer may be unclear as to whether the software will behave in a predictable manner after a change is made or whether conflict will occur with other developers when a commit is made. Such uncertainty arises because other de-

velopers would have committed changes that the developer would not know of until he/she attempts to commit his changes.

We also explained in Sec. 2.2.4.3 how a modular software structure could reduce the uncertainty related to the development process by limiting the dependencies between modules, thereby enabling parallel development by contributors (Baldwin and Clark, 2006) and reducing the maintenance cost of any introduced bugs (Page-Jones, 1998). In addition, similar functionality is encapsulated into modules that reduce the cognitive burden on potential contributors to acquire complete knowledge. We mentioned also that these benefits are contingent upon the stability of the source code structure (Darcy et al., 2005). Given that the software is an explicit form of the requisite development knowledge, we argue that the degree to which development knowledge is incomplete or uncertain could be estimated by the magnitude of change in modularity of the code base.

Procedure

The process in which we obtained a modularity measure is summarized below:

1. Extract dependency graph of source code between source files.
2. Perform leading eigen-vector method partitioning that maximizes modularity (See Appendix C).
3. Extract the modularity measure from the partitioned graph along with the number of modules.

We extracted the modularity value for both the beginning and end of the analysis period and used the difference to estimate the change in the structure of the code base, which we

then used as a proxy for knowledge completeness. Since change in modularity measure can be either positive or negative, then the magnitude of change will be smallest close zero and highest close to the maximum and minimum values. As such, knowledge completeness will be highest for low magnitudes of change whereas it will be lowest for high magnitudes of change. Because we are interested in the effect of the magnitude of change in modularity and hypothesizing that the effect of low magnitudes of change differ from that of higher magnitudes, then it is best to test for this effect using non-linear terms. Specifically, it is best tested using a cubic effects because it can capture the effect of positive change in magnitude independently of negative change.

3.2.4 Knowledge Diversity: Amount

Content Validity

According to our discussion in Sec. 2.2.4.4, knowledge amount refers to the number of distinct knowledge domains from which the requisite development knowledge to contribute to a FLOSS community draws. Given that software is an explicit form of this knowledge, we could determine its amount by identifying the number of unique modules in the code base. Assuming FLOSS communities are effective in modularizing source code, which they are (MacCormack et al., 2006), the result is that highly related programming tasks are grouped in a single cohesive module (Page-Jones, 1998). The functionality within a cohesive module will draw from the same body of knowledge. This knowledge will differ from the knowledge contained in other modules. Therefore, we argue that

knowledge amount could be estimated using the number of cohesive functional units (i.e., modules) in the code base.

Procedure

Since the leading eigen-vector method (See Section 2.3.2.3) identifies modules by maximizing cohesion and minimizing coupling, we used the number of identified modules from this method at the beginning of the analysis period as an estimate for the amount of requisite development knowledge.

3.2.5 Knowledge Diversity: Relatedness

Content Validity

According to our discussion in Sec. 2.2.4.4, knowledge relatedness refers to the extent to which the different knowledge domains from which the requisite development knowledge of a FLOSS community draws are related. Related knowledge is difficult to split and must be absorbed as a whole into an individual's mind in order to be of use (Galunic and Rodan 1998). When knowledge is highly unrelated, the different knowledge domain can easily be identified, and the modules of the code base clearly reflect them. As knowledge relatedness increases, knowledge from different domains becomes more difficult to distinguish or separate. As a result, the modules reflecting such knowledge would draw from more knowledge domains, resulting in larger source code modules.

Procedure

Once the number of modules using the leading eigen-vector method (See Section 2.3.2.3) is identified, we divided the total lines of source code by it. The result is the average size of the module measured in source lines of code.

3.3 Controls

3.3.1 Number of Committers

Seeking and finding the right knowledge is an important step in knowledge acquisition (Alavi and Leidner, 2001). Whether this knowledge is tacit or codified, finding knowledge that is relevant to a developer's needs might prove to be a formidable task. In active FLOSS projects, knowledge related to where relevant documentation resides is already internalized by the developers who contribute frequently, such as committers. This allows the committers to assist other developers and makes it easier for them to find relevant knowledge, thereby reducing the effort needed to make a contribution.

Furthermore, there is evidence that committers can be overloaded from community interaction, which could limit the amount of knowledge transferred from committers to contributors (Kuk, 2006). Communities with a greater number of committers will have a greater capacity to transfer development-related knowledge, and since communities observe a great degree of variability in their number of committers, it becomes necessary to control our results for their numbers. Given that the revision control system lists the name of the committer that performed the for every commit performed, we extract these names and normalize them for the analysis period (See Appendix B). We count the unique normalized names of committers for the analysis period and use it as a control.

3.3.2 Popularity of the FLOSS project

The popularity of a FLOSS project might also play a role in the number of contributors simply because contributors are themselves users. Therefore, a FLOSS community's numbers are limited by how many people know of and use its software. Furthermore, assuming that projects with higher value will attract a greater number of users (c.f. Raymond, 2001;Lerner and Tirole, 2002; Roberts et al., 2006; Lakhani and Wolf, 2007), popularity can serve as a control for the effect of value on participation. This would allow us to capture the effect of reducing cost of participation that results in the increase in number of contributors that is independent of the effect of value.

To control for popularity, we used the number of users that reported using the software on ohloh.net. Given that ohloh.net is a social website for FLOSS developers, the number of users that report using a software product serves as a good proxy for popularity amongst developers.

3.3.3 License Restrictiveness

Following the work of Stewart et al. (2006), we operationalized license restrictiveness by coding the licenses based on whether they included the General Public License (GPL) or not. GPL is one of the most restrictive FLOSS licenses with provisions that prevent the mixing of FLOSS source code with proprietary source code and the requirement that derivative work be released under the GPL license. Such provisions might prevent many users, especially for-profit organizations, from leveraging FLOSS for their own use. This

could limit a FLOSS community's number of users and, as a result, the number of potential developers.

3.3.4 Project Age and Analysis Period

The age of the project is an important control that serves as a proxy for several other factors (Stewart et al., 2006). For example, older projects could be well established and more popular than newer projects. Project age may also serve as a proxy for the experience or familiarity the committers have with the community, which could make their interactions and knowledge exchanges easier. We calculated the age of each FLOSS project as the number of weeks from the first commit to January 1, 2007 with projects that have a negative age signifying that they began after Jan 1, 2007. In addition, we control for the analysis period from the start of the project. This is important since projects might undergo different stages of growth, which could influence the number of new contributors per stage.

3.3.5 Programming Language

The programming language used for each FLOSS project could also have an impact since it is inherently easier to make changes to scripted languages, such as Python, than it is to compiled languages, such as C, which require the mastery of a different set of building and compilation tools. In addition, code written in Python is inherently shorter than code written in C or C++ given the dynamic nature of the language. The popularity of the programming language among developers will also limit the pool of potential developers that

might participate in a FLOSS project. We used dummy variables to encode the languages used in the code bases, since all three could be mixed together.

3.4 Analysis and Results

Given the longitudinal nature of our data, we used mixed models where the project is a level two random effect variable. In addition, we modeled our main effect variables are as level one random effects which takes into account the serial correlation between observations (Cohen et al., 2003). The first step we performed in our analysis was to assess the distributional characteristics of our variables to ensure that none of the main assumptions of mixed model analysis were violated. Since most our variables are count based, we either had to perform log or negative power transformations to ensure that our independent variables did not violate the assumption that the variables were normally distributed. As for the dependent variable, since it is a count-based variable with no negative values, we opted to fit our model using a poisson-based generalized mixed model (Gardner et al., 1995; Cohen et al., 2003). As a result, we checked for signs of any violations to the assumptions of a poisson-based regression model. Specifically, we looked for signs of zero inflation or over-dispersion and found our data to satisfy the requirements of the model with less than 25% of the observations of the dependent variable being zeros and the dispersion parameter showing a value close to one[14] (Gardner et al., 1995; Warton, 2005).

[14] We performed the additional step of fitting a QuasiPoisson generalized mixed model to deal with the case that over dispersion might exist in our data (Gardner et al., 1995) and ended up with the exact same results as a poisson model including inference tests. The dispersion parameters for the different models we fitted were between the values .6 & 1.7.

The next step before the analysis was to assess the correlations of our variables to determine the discriminant validity of our variables and to discount any issues in our analysis related to multicolinearity. Table 2.8 provides the correlation between the variables used in our study and some descriptive statistics. Given that there are no unusually high correlations between the variables, we concluded that our variables are distinct and proceeded to check if there were any projects that exerted an unusually high influence on the result of our analysis. Using Cook's distance (Cohen et al., 2003) from the influence.ME package (Nieuwenhuis et al., 2009), only a single project exhibited an unusually high influence with a Cook's d value of one (Cohen et al., 2002). The observations from that project were excluded from the analysis, which brought down the number of analyzed projects to 234 and total observations to 1823.

The subsequent step was to fit our statistical model and to make some statistical inferences; for this step we used the R statistical package version 2.9.1 (R Development Core Team, 2009) in addition to the lme4 library for fitting random effect models (Bates and Maechler, 2009). By process of adding our main effect variables and assessing the significance of including the variable as a random effect using Chi^2 difference tests, we identified KAMT, KREL, KCOD, and KCOMP as random effects embedded within projects.

Table 2.8: Variable correlations and descriptive statistics

	SUS	KCOD	KCOMP	KREL	KAMT	COM	AGE	PER	isGPL	isC	isCpp	isPy	POP
SUS	1.00												
KCOD	-0.10	1.00											
KCOMP	-0.02	0.02	1.00										
KREL	0.19	-0.32	-0.05	1.00									
KAMT	0.30	0.00	0.03	-0.07	1.00								
COM	0.40	-0.12	0.00	0.13	0.38	1.00							
AGE	0.15	-0.04	-0.03	0.30	0.13	0.12	1.00						
PER	-0.06	0.00	-0.02	0.03	0.05	-0.01	0.04	1.00					
isGPL	0.02	-0.05	-0.02	-0.04	0.00	0.04	0.09	0.05	1.00				
isC	0.18	-0.34	0.01	0.38	0.09	0.18	0.33	0.00	0.10	1.00			
isCpp	-0.01	-0.13	0.02	-0.02	0.40	0.15	-0.09	-0.02	0.09	0.01	1.00		
isPy	-0.09	0.41	-0.02	-0.35	-0.14	-0.07	-0.27	0.00	-0.14	-0.58	-0.33	1.00	
POP	-0.34	0.02	0.03	-0.18	-0.18	-0.24	-0.40	-0.03	-0.12	-0.20	0.07	0.20	1.00
min	0.00	-0.37	-0.45	-6.39	-3.44	-1.73	-7.63	1.00	0.00	0.00	0.00	0.00	-0.21
mean	5.96	0.00	0.00	0.00	0.00	0.00	0.00	4.54	0.42	0.82	0.43	0.14	0.00
median	3.00	0.00	0.00	0.04	-0.11	-0.12	0.26	4.00	0.00	1.00	0.00	0.00	0.00
max	94.00	0.56	0.55	4.77	3.63	3.09	13.14	10.00	1.00	1.00	1.00	1.00	0.15
std	4.45	0.09	0.01	0.93	1.03	1.36	2.59	2.97	0.00	0.00	0.00	0.00	0.11

Table 2.9 summarizes the results of our model fitting statistical analysis. The first of these models is the null model used as a baseline to determine the explanatory power of subsequent models. The second model is the control only model for which we included the control variables as fixed effects. As can be seen from Table 2.9, the control model has an R^2 of %14.6 based on the reduction in the deviance value from the null model. The Chi^2 difference test also suggests that the control model explains significantly more variance than the null model.

Table 2.9: Model fitting

Model	Null	Control			Main Effects			Quadratic			Cubic		
		Estimate	Std. Err.	*p*	Estimate	Std. Err.	*p*	Estimate	Std. Err.	*p*	Estimate	Std. Err.	*p*
$KCOMP_3$											-14.242	5.643	0.012*
$KCOMP_2$								-3.043	1.538	0.048*	0.28	2.01	0.889
$KCOD^2$	—							-9.478	4.133	0.022*	-9.443	4.149	0.023*
$KAMT^2$	—							-0.082	0.029	0.005**	-0.082	0.029	0.005**
$KREL^2$	—							-0.12	0.036	<.001***	cubic	0.036	<.001***
KCOD	—				-0.896	0.975	0.363	-0.693	0.947	0.464	-0.715	0.949	0.451
KCOMP	—				1.580	0.531	0.003**	1.626	0.549	0.003**	1.994	0.567	<.001***
KAMT	—				-0.060	0.079	0.525	-0.04	0.08	0.617	-0.045	0.08	0.575
KREL	—				0.157	0.095	0.123	0.124	0.096	0.193	0.129	0.096	0.176
COM	—	0.731	0.030	<.0001***	0.805	0.034	<.0001***	0.791	0.034	<.001***	0.789	0.034	<.001***
AGE	—	0.008	0.032	0.806	-0.011	0.039	0.778	-0.018	0.038	0.642	-0.014	0.038	0.694
PER	—	-0.031	0.004	<.0001***	-0.032	0.005	<.0001***	-0.031	0.005	<.001***	-0.031	0.005	<.001***
isGPL	—	0.033	0.153	0.832	-0.027	0.175	0.921	-0.058	0.171	0.734	-0.057	0.171	0.738
isC	—	0.471	0.252	0.061.	0.340	0.304	0.262	0.274	0.294	0.352	0.254	0.294	0.388
isCpp	—	-0.286	0.167	0.086.	-0.268	0.198	0.192	-0.296	0.192	0.123	-0.28	0.193	0.146
isPy	—	0.005	0.291	0.97	0.36	0.358	0.319	0.35	0.341	0.305	0.342	0.342	0.317
POP	—	2.074	0.890	0.02*	1.992	1.037	0.055.	1.857	1.009	0.066.	1.818	1.013	0.073.
Int	1.009***	0.849	0.275	0.002**	0.974	0.338	0.003**	1.07	0.319	<.001***	1.085	0.32	<.001***
AIC	5619		4812			4574			4549			4545	
LogLik	-2807		-2396			-2259			-2242			-2239	
Deviance	5615		4792			4518			4485			4479	
R^2	—		%14.6			%19.53			%20.13			%20.24	
Chi^2 *Diff*	—		822.468			274.282			29.639			6.037	
p	—		<.0001***			<.0001***			<.0001***			0.014*	

Significance codes: '***' <0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Our interest lies mostly with the main effects model in which the Chi^2 difference test suggests that the model explains significantly more variance than the control model with an R^2 of %19.53. Among the coefficients of the hypothesized effects, only KCOMP is found to be significant at p=0.003 and estimated at 1.58. However, since we are interested in effect of magnitude of change, the linear effect for KCOMP is not of interest to us.

As for the non-supported effects, we consider if the reason for obtaining such a result was due to the non-linear nature of these effects. If you recall from Sec. 2.2.1.2, we explained how value of the software or community benefits could also play a role the decision of value-maximizing developers to make a contribution. The interplay between such value and contribution cost that grows with further software development (Lehman et al., 1997) gives us reason to believe in the existence of non-linear relationships between these variables and sustainability. In addition, we made the assumption that value is distributed within a narrow range across FLOSS communities for a linear effect to be detected. The results so far suggests the violation of this assumption, however, we would like to know if this holds true for all FLOSS communities or a subset of them. More importantly, we would like to know if this is a result of the non-linear nature of their relationship or that there is no independent effect for cost.

To search for the non-linear effects, we continued our investigation by testing for quadratic relationships. The expectation is that high levels of our variables of interest will have different value and cost implications on a FLOSS community than low values of the same variables. For example, a community with might have a small code base because the software is at the initial stages of development and is yet to be of value for many users. Adding features for communities at this stage might yield higher value than cost and would result in effects that are opposite to what we predicted.

To test for quadratic effects, we added each quadratic term individually and used a Chi^2 difference test to ensure that each term explained a significant amount of variability. Our investigation resulted in the higher order model (depicted in Table 2.9) in which

KAMT, KREL, and KCOD all observed negative and significant quadratic effects. Furthermore, the Chi^2 difference test suggests that the higher order model explains a significant amount of variance above and beyond the main effects model with an R^2 of %20.13. To get a better understanding of the nature of these relationships, we graphed in Figures 2.4-2.9 the curves representing the quadratic relationship followed by the graph of the simple slopes to illustrate how the relationship changes over different values of the KAMT, KREL, and KCOD.

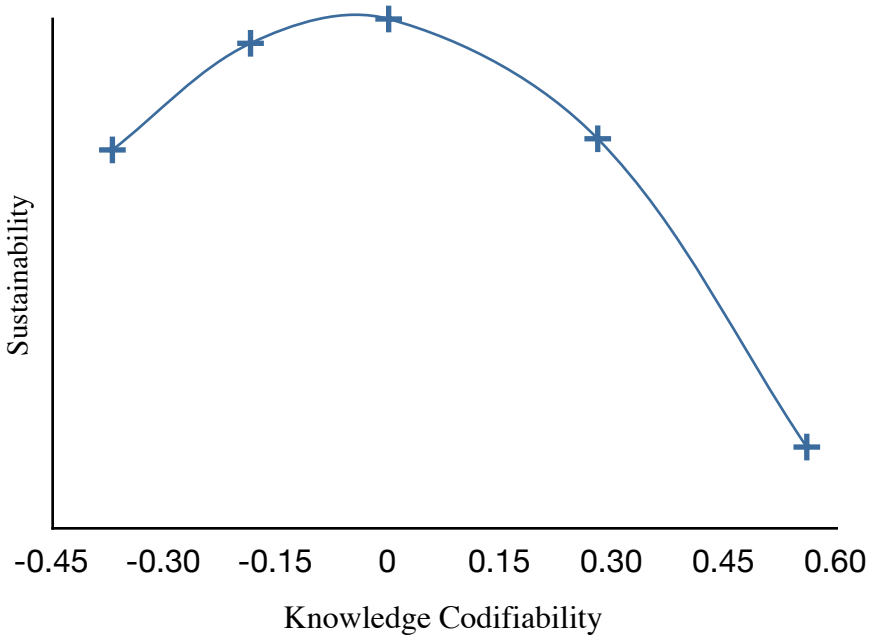

Figure 2.4: Quadratic effect of knowledge codifiability

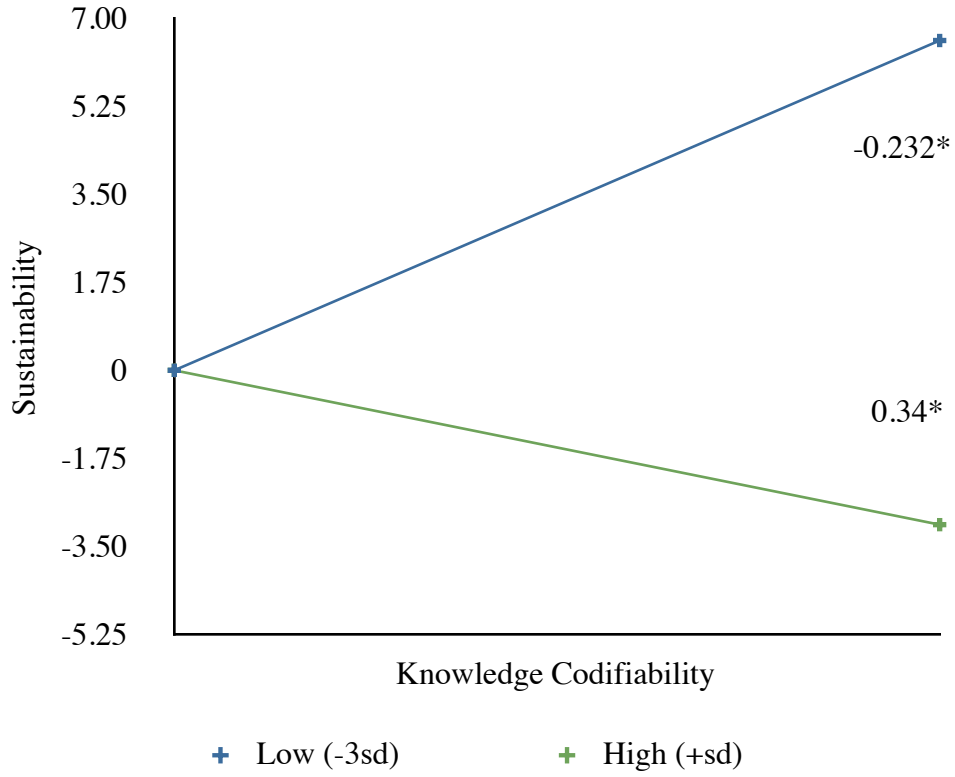

Figure 2.5: Simple slopes for the quadratic effect of knowledge codifiability

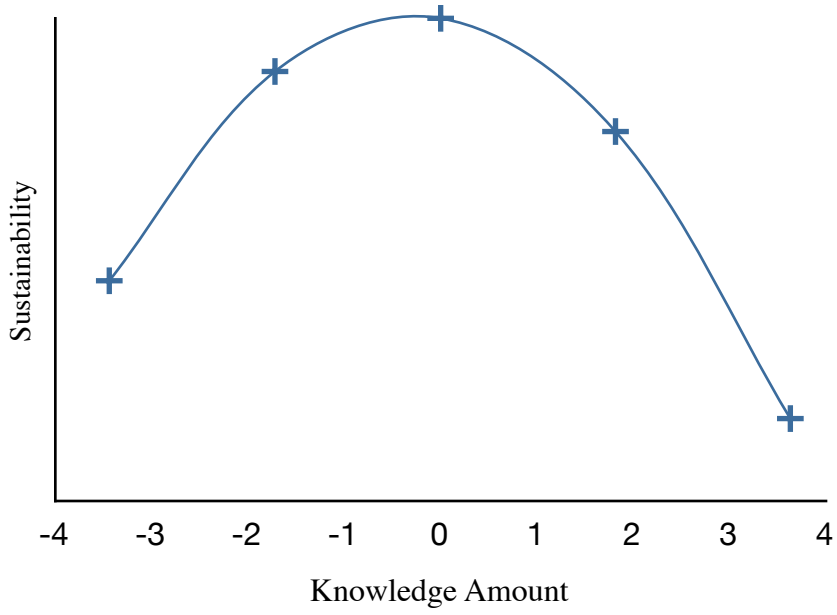

Figure 2.6: Quadratic effect of knowledge amount

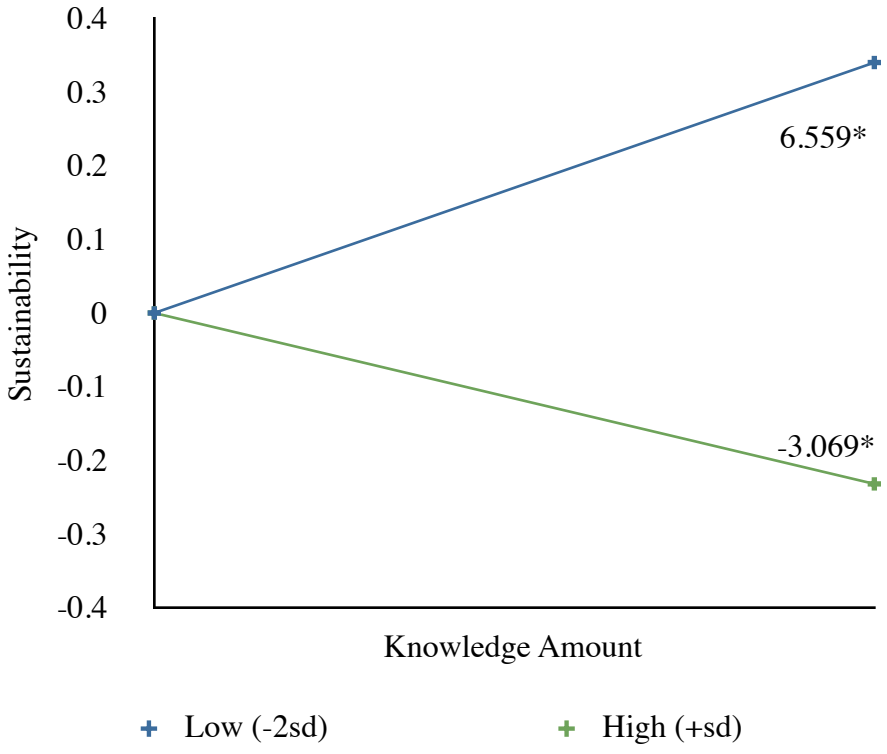

Figure 2.7: Simple slopes for the quadratic effect of knowledge amount

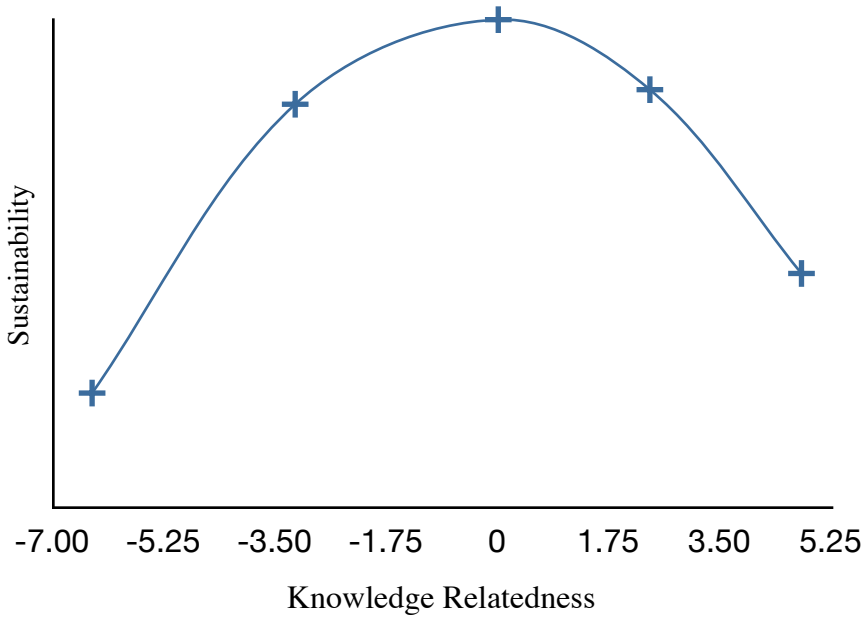

Figure 2.8: Quadratic effect of knowledge relatedness

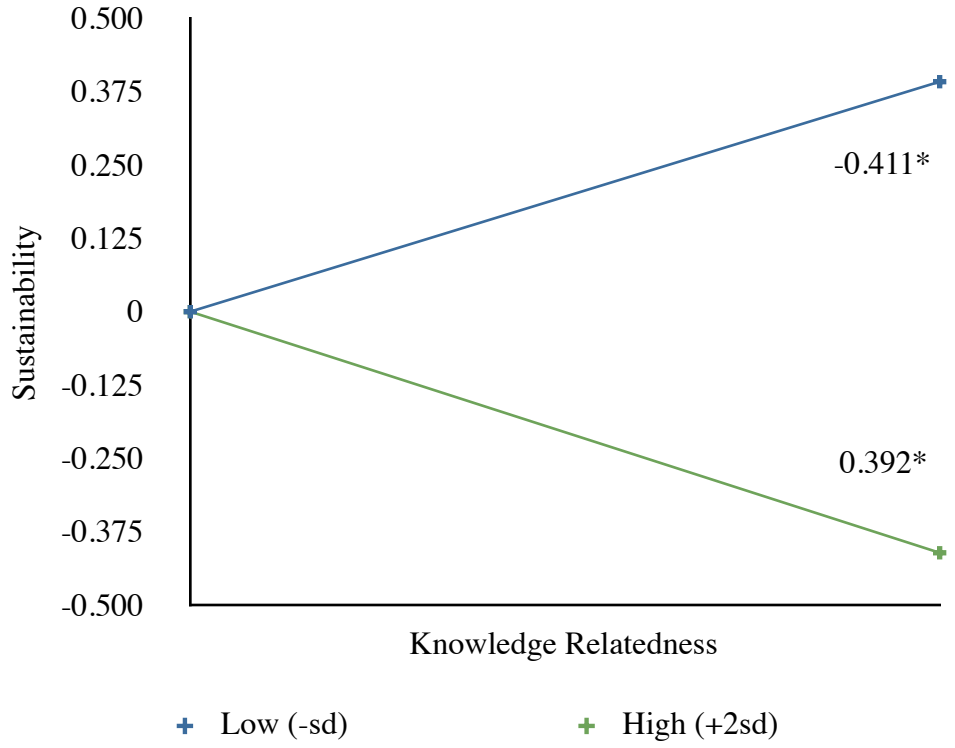

Figure 2.9: Simple slopes for the quadratic effect of knowledge relatedness

The quadratic effects in figures 2.4, 2.6, and 2.8 suggest that the nature of the relationship to sustainability is changing for different values of the independent variables in question. To illustrate this, we plot the simple slopes in figures 2.5, 2.7, and 2.9. Moving in single standard deviation steps away from the mean, we plotted the first simple slope to exhibit significance based on a two tailed t-test and the following value for the standard error:

$$S_b = \sqrt{S_{11} + 4XS_{12} + 4X^2S_{22}}$$

where S_{11} and S_{22} are variance in the regression coefficients for the main and quadratic effects respectively, and S_{12} is the covariance between these same coefficients (Aiken and West, 1991).

As can be inferred from the simple slope graphs, the negative relationship hypothesized in H1 is supported for below average values of KCOD. The support is evident from the significant and negative slopes in Figure 2.5 for values standard deviations or more below the mean. This might seem unlikely to occur, however, the absolute minimum value for KCOD is zero when there is no documentation in the code base. The significant simple slope for values below three standard deviations suggests that having some documentation is certainty better than none. H2 and H4 are supported for the above average and high values of both KAMT and KREL. This is evident from the significant and negative simple slope values in Figures 2.7 and 2.9 for values that two and one standard deviations above the mean respectively.

Notice that we also obtained a significant and negative effect for the coefficient of the quadratic KCOMP term (-3.043, p-value = 0.048). While this term suggests that there is a detrimental effect for any change in the absolute value of the modularity measure, the quadratic term will not tell us if the smaller changes will result in a positive relationship between knowledge completeness and sustainability. To uncover such an effect, we test for a cubic KCOMP coefficient and find it to be negative and significant (-14.242, p-value = 0.012). We depict this cubic effect in Figure 2.10.

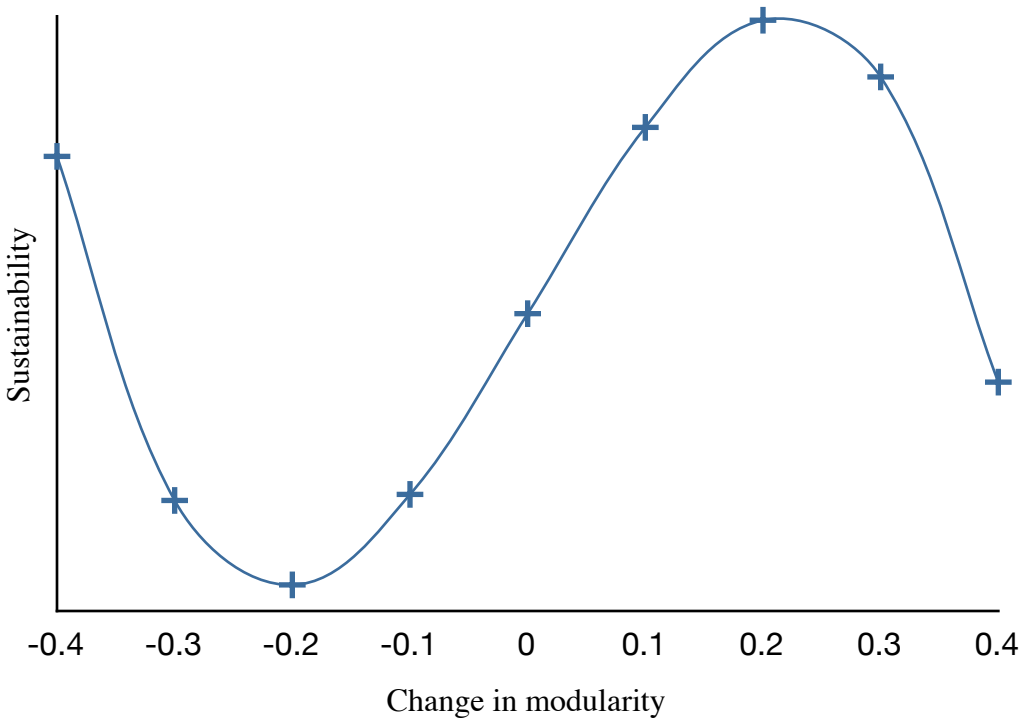

Figure 2.10: Cubic effect of knowledge completeness

To simplify interpretation, we plot two simple slope graphs for the cubic effect of KCOMP. The first, depicted in Figure 2.11, is a graph representing positive change in the KCOMP measure. Given that KCOMP is mean centered, notice how the coefficient of the simple slope for the line depicting small change is positive and significant (1.994, p-value = <0.001). The simple slope shares the same significance test as the one for the linear effect of KCOMP in the cubic model (Aiken and West, 1991). This line suggests that the relationship between knowledge completeness and sustainability is positive for small changes in the structure of the code base (i.e., high knowledge completeness) supporting H2 for high levels of knowledge completeness. Notice also how the relationship between sustainability and knowledge completeness reverses its direction for high values of change in modularity measure (i.e., low knowledge completeness). The change in the

slope is significant as indicated by the significance of the cubic KCOMP term, supporting H2 for low levels of knowledge completeness.

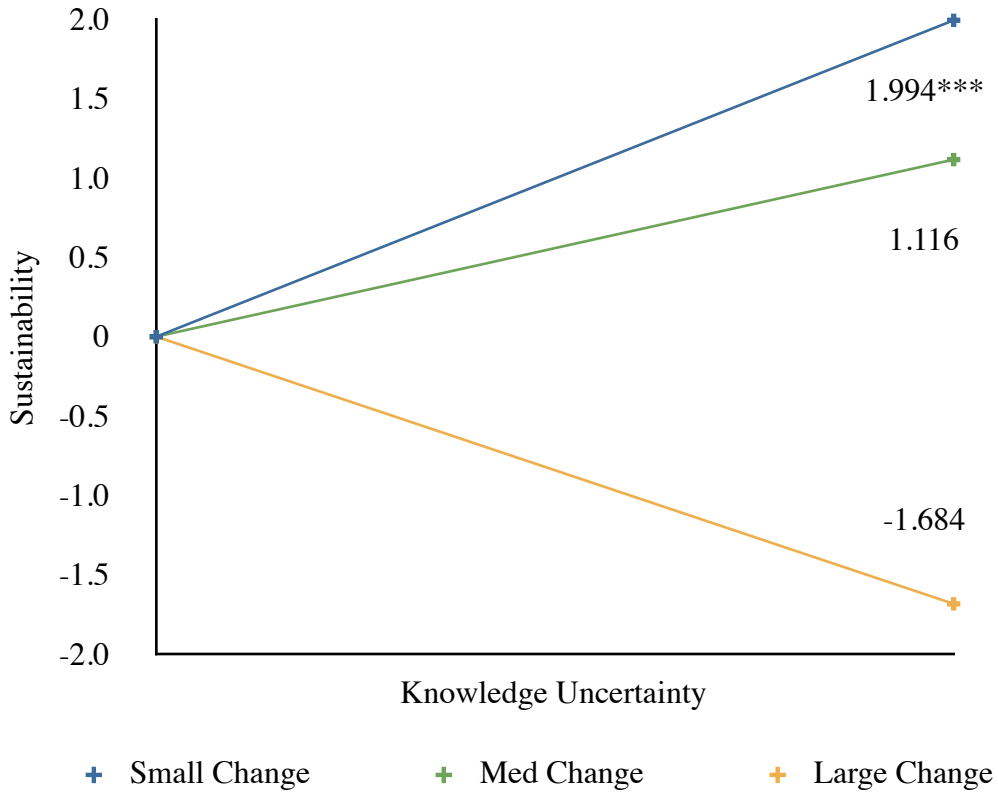

Figure 2.11: Simple slopes for positive change in knowledge completeness

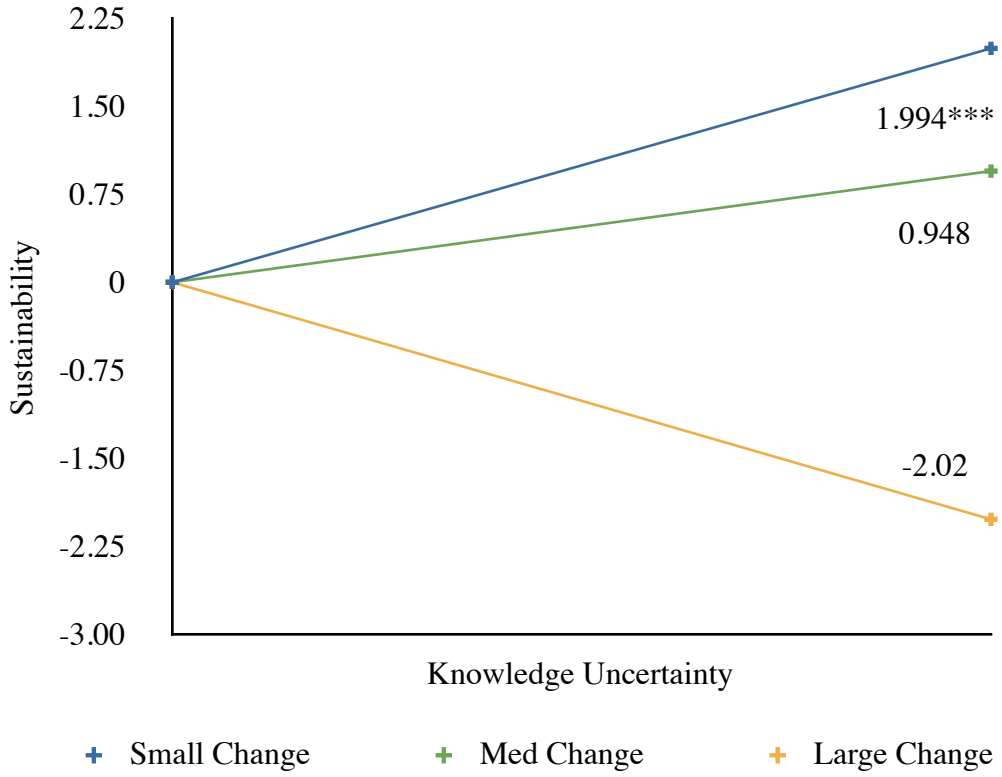

Figure 2.12: Simple slopes for negative change in knowledge completeness

We also plot the simple slopes for the effect of negative changes in the modularity measure in Figure 2.12. As can be seen, the results for negative changes in modularity measure mirror those of the positive changes. With a positive and significant simple slope for knowledge completeness for low levels of change in modularity measure (1.994, p-value = <0.001), the relationship between sustainability and knowledge completeness is positive for low levels of knowledge completeness. The relationship reverses itself for more extreme values of negative change in modularity.

As for the small effect sizes for the cubic and quadratic models, it is typical when looking for higher order effects in archival data (e.g., Kuk, 2006; Lavie, 2007; Gulati et

al., 2009) and inline with the 0.002 median effect size for moderation effects found by Aguinis et al. (2005) in their review of management and psychological literature from the past 30 years. Both the error present in the data and amount of residual variance after partialling out the main effects make it difficult to detect higher order effect (McClelland and Judd, 1993). Furthermore, higher order effects with small magnitudes will be even more difficult to detect and will require large sample sizes (Ullrich et al., 2008; Champoux and Peters, 1987). The fact that we found a significant effect for our higher order effects given all these odds against finding them suggests that the actual effect might be even greater than what the results are telling us. Furthermore, Champoux and Peters (1987) suggest that the significance and magnitude of the effect are more important than the change in effect size in determining the importance of the higher order effect.

4 Discussion

For the purpose of discussing our findings, we summarize our empirical findings in Table 2.10. We hypothesized in H1 that knowledge codifiability will have a positive relationship to sustainability, which is defined as a FLOSS community's ability to convert users to contributors effectively in order to maintain development efforts. We argued that making requisite knowledge more explicit would reduce contribution costs by making such knowledge more accessible. What we found is that this hypothesis holds for below average values of knowledge codifiability and starts to reverse in direction for above average values of knowledge codifiability. We provide an explanation of this reversal in the direction of the relationship in the following section.

Table 2.10: Summary of empirical findings from the higher order model

Hypothesis	Coeffecient	Support
H1: The extent to which source-code-related knowledge for a FLOSS community is codified will be positively related to the sustainability of that community.	$KCOD^2$: -9.954 p=0.016*	Supported for below average values of knowledge codifiability.
H2: The extent to which source-code-related knowledge for a FLOSS community is complete will be positively related to the sustainability of that community.	$KCOMP^3$: -14.242, p=0.012**	Supported.
H3: The amount of source-code-related knowledge that exists in a FLOSS community is negatively related to the sustainability of that community.	$KAMT^2$: -0.088 p=0.002**	Supported for above average values of knowledge amount.
H4: The relatedness of source-code-related knowledge that exists in a FLOSS community is negatively related to the sustainability of that community.	$KREL^2$: -0.129 p=0.0003***	Supported for above average values of knowledge relatedness.

We argued in H2 for a positive relationship between knowledge completeness, which was operationalized as the change in modularity of the code base and sustainability. We argued that a a stable code base code base would convey more complete knowledge for a contributor that requires to integrate a patch into the community code base. Complete knowledge entails that ones the code base is checked out, the contributor would not be required to to rewrite his patch at time of committal because the community code base changed significantly in its structure rendering his/her patch incompatible. As a result, the effort to integrate a patch is reduced, thereby, increasing the number of new contributors that are likely to participate. H2 was best supported in the cubic model that shows a positive relationship between knowledge completeness and sustainability for high levels of

knowledge completeness. This relationship reverses direction when for extreme values of knowledge completeness whether they are positive or negative.

There are a number of important implications for both theory and practice from the results of H2. The finding provides empirical support for theories that propose that source code structures could encourage participation (Baldwin and Clark, 2006). The findings also lend support to the validity of the modularity measure since the measure observed behavior that we predicted prior to knowing the results of the analysis. As for practice, this finding highlights the importance of stability of code design and the importance of the release early release often practice. Such a practice encourages small incremental changes that do not bring significant changes to the community code base, which our work suggests is important to encourage participation from new contributors.

In H3 and H4, we argued that both dimension of diversity (i.e., amount and relatedness) for knowledge that exits in a FLOSS community are negatively related to the sustainability of the FLOSS community because they increase contribution costs. This relationship was found to hold for above average values of knowledge amount and relatedness, as can been seen from the significant and negative slopes in Figures 2.5 and 2.6. This negative relationship was attributed to the increased contribution cost associated with the increased effort in comprehending a code base that consists of a diverse and large number of modules. While we have discussed the benefits of modularity in reducing contribution costs by allowing developers to contribute by working and comprehending individual modules, knowledge relatedness negates the effect of modularity by requiring a developer to comprehend a group of highly related modules. The result is that develop-

ers are required to expend greater effort to acquire knowledge that is required to make the initial contribution.

These results highlight how certain projects are inherently complex and are likely to be unsustainable in the long run. The inherent complexity highlights the need for FLOSS communities to be clear in terms of the goals they want to achieve with the software products they are developing. An effective way of managing complexity is to partition complex projects into smaller related projects that deal with specific, but smaller, problems. Such an approach is only enabled by open standards. An example of this strategy is evident in the Python community in which the WSGI protocol is a standard that allows different applications to work with web-requests and pass it on to the next application that could make use of it. This allowed a highly complex web framework known as ZOPE to be partitioned into smaller, but manageable, FLOSS projects[15].

Furthermore, the complexity highlights one of the strengths of the FLOSS marketplace: renewal. FLOSS projects could grow to be complex simply from the accumulation of source code over time (Lehman et al., 1997). Such projects become difficult to maintain and might spur potential contributors who cannot break into a community to develop their own FLOSS solutions that might surpass older and more complex solutions that they originally considered using, thereby, heightening the competition in the FLOSS marketplace. This is highlighted by the emergence of the Django web framework to replace the Zope web framework, and now the emergence of micro web frameworks to replace Django, which itself is becoming difficult to maintain.

---

[15] See http://repoze.org/

4.1 Post-Hoc Analysis of Quadratic Effects

The quadratic effects are very interesting because the relationships' directions are reversed. This warrants a closer examination of the unexpected reverse effects in light of the FLOSS marketplace framework that we have developed. We start with the negative effect associated with an increase in knowledge codifiability as measured by the ratio of lines of documentation to the lines of code. This negative relationship becomes evident and significant for above average ratios of documentation to code. This is the least surprising of the reverse effects, given that we made the assumption that contributors are boundedly rational (Simon, 1955). As the amount of documentation increases, potential contributors are still required to expend effort to read the documentation and find their way through it to get to the relevant parts. Therefore, the effort to acquire the knowledge embedded in the documentation might increase and may reach a point where the value the contributor will get from contribution no longer justifies the effort required to read the documentation, let alone make the contribution.

This finding has a very important implication to FLOSS communities, as it suggests that long-known truths about the value of information still hold in the new age of distributed software development (Ackoff, 1967). The increased amount of documentation might be a signal that the quality of the documentation is poor or that too much unnecessary information is included, making it more difficult to find relevant knowledge. Therefore, FLOSS communities should be aware of how critical documentation is. Even though documentation is often considered a chore and many developers tend to avoid it

(Lakhani and Wolf, 2007), the quadratic effect suggests that it is necessary and, more importantly, that it must be done right or it could have the reverse effect on potential contributors (at the minimum). FLOSS communities may want to consider hiring or soliciting the assistance of individuals with good technical writing skills to edit and improve the documentation. Furthermore, the documentation should be continuously improved just like the software itself and not be left to accumulate unnecessary or outdated bits of information over time.

The positive relationships for low levels of knowledge amount and relatedness were the most surprising of our empirical findings. While it is contrary to the cost argument we used to explain how effective FLOSS communities convert users into contributors, it highlights another very important dynamic in the FLOSS marketplace. We are referring here to the potential value that is generated from implementing and adding source code. Needless to say, adding such source code would result in an increase in complexity, but this is the case with any conventional software project (Lehman et al., 1997), especially successful projects. The added value would result in an increase in the number of users of the software, which might indirectly result in a greater number of new contributors.

Furthermore, the unexpected positive relationship could highlight the importance of implementing just enough of the code base to allow developers to gravitate around it (AlMarzouq et al., 2005). With a significant portion of the source code written that offers just enough functionality to be useful, potential contributors can leverage all of the knowledge behind the software without needing to acquire the development knowledge fully and re-implement what has already been implemented in the code base. As knowl-

edge diversity from implementing more functionality increases, the cost of internalizing all of the underlying knowledge requirements also increases. When a software project that requires diverse knowledge is released as FLOSS, the rational choice for any developer that seeks to contribute to such a project is to pool his/her effort with that of the community. This way, the developers can specialize in creating specific components rather than working on the whole code base, resulting in higher quality and more innovative software (Grant, 1996a).

However, it is important to note that the range of values for which the effect of knowledge diversity starts to reverse its effect cannot be specified ex-ante. The reason for this is that these values have to be taken into consideration relative to other characteristics of the source code and of the FLOSS project, as our statistical analysis suggests. Because each project is unique in its design and the characteristics of its underlying knowledge, there is no specific number of modules or module size that fits all. More importantly, the complexity of the code base cannot be known until the source code is actually written. Even if it is known, FLOSS communities sometimes consciously choose to make the software complex in order to maintain compatibility with older versions of the software that might be used by community members or interoperate with other software packages. This finding highlights the complexity of the trade-offs that have to be made by FLOSS communities.

Therefore, the quadratic effect related to the dimensions of knowledge diversity suggests that when a developer is faced with the choice between working with a FLOSS community or repeating the effort of the community by developing a similar solution us-

ing a proprietary model, the rational choice would be to pool the developer's effort with that of the FLOSS community. However, there is only a certain amount of complexity that the developer can tolerate; otherwise, the rational choice might be to simply create a new project or develop the software solution in-house. Similar to the implications related to H3 and H4, this finding highlights the importance of the renewal of complex projects in the FLOSS marketplace. Furthermore, it suggests that such acts of project renewal may have a good probability of success when they are released as FLOSS. The main reason for this success is that the alternatives to participating a highly complex FLOSS project or developing a similar propriety system might be less cost effective.

Another important effect of the quadratic effects of the dimension of knowledge diversity is that the success of a FLOSS community, as measured by the number of new contributors joining the development effort, is a delicate balancing act between what features should be included in the code base and how much complexity such features introduce. The results suggest that an increase in complexity is healthy to a certain degree, as it adds value to users and contributors. Adding too many features, however, could be detrimental to the sustainability of the project in the long run. Adding too many features might be a result of the community's lack of clear goals as to what problems they would like their software to target. Therefore, it is important for communities to set goals such that it becomes clear to committers when to refuse and when to incorporate contributed features into the code base.

5 Brooks' Law Revisited

According to Brooks (1975), one of the main causes for software project delays is due to the current developers' distraction from training newly added members, which is referred to as the ramp-up effect. It has been debated in the research community whether the Brooks' contributions, collectively known as Brook's Law, apply in the FLOSS development context (Capiluppi and Adams, 2009) or not (Koch, 2004; Schweik et al., 2008). The debate mostly focused on the effort needed to communicate but disregarded the effects of a ramp-up. Our study's findings contribute to this discourse by proposing a conceptualization of the ramp-up effect fit for the FLOSS context and by providing empirical support of its effects.

One way in which FLOSS development differs from conventional software development is that developers are volunteers (Raymond, 2001). As a result, members are under no obligation to waste their time in training other contributors. However, the knowledge embedded in the source code itself allows potential contributors to bring themselves up to speed by reading the source code and acquiring the necessary knowledge to make contributions (Raymond, 2001). Since new contributors do not significantly distract current developers, it is assumed that the ramp-up effect is not relevant to FLOSS development. However, when the ramp-up effect is conceptualized as the knowledge barriers that a developer must overcome to become an effective team members, it becomes easier to argue for the ramp-up effect's influence in the FLOSS context.

If the ramp-up effect is viewed as a knowledge barrier, it implies that it varies from one project to the next (Attewell, 1992). Therefore, in conventional teams, the amount of effort from or the distraction to current members will be proportional to the size of these

effects (Attewell, 1992). However, in FLOSS teams, the size of the ramp-up effect will be proportional to the effort expended by potential contributors in reading the source code and acquiring the embedded knowledge. Therefore, we argue that projects that have high knowledge barriers due to the nature of the knowledge embedded into the code base will deter potential contributors from joining the development effort. The result of such situations is that the FLOSS project would lose effort that could have potentially been gained if the knowledge barriers were lower. Furthermore, these knowledge barriers might impact the propensity of the FLOSS project to sustain development over time.

Our empirical findings support the idea that knowledge barriers and, therefore, the ramp-up effect vary between projects. The magnitude of these barriers will depend on the codifiability, completeness, and diversity of the knowledge underlying the source code. Therefore, we conclude that the ramp-up effect is indeed still relevant in the FLOSS development context. However, because participation in FLOSS teams is voluntary, the ramp-up effect impacts the performance of FLOSS communities through different dynamics than in conventional software development teams.

Given that software production is considered a knowledge intensive task, other online and virtual communities focusing on the production and dissemination of knowledge could also benefits from our findings. Communities of practice and knowledge production communities, such as Wikipedia, that rely on the participation of volunteers could benefit from increasing the numbers of new contributors by lowering the contribution costs. As with FLOSS communities, an initial contribution to Wikipedia will require the participant to learn and comprehend the article that he/she is about to contribute to, understand how

to use the tools to contribute, and coordinate with the gatekeepers of the community to have their contribution accepted. Future research on such communities could benefit from our conceptualization of the ramp-up effect as a bases to understand the costs associated with the initial contribution.

6 Limitations and Future research

Although we have built up the logic to argue that the reduction of transaction costs is the most effective strategy for increasing the number of external participants, we believe that the value appropriated by individuals from contributing to a FLOSS project is still valuable to understand contribution behavior, especially at the individual level. Our novel approach in framing the FLOSS marketplace opens up the possibility for leveraging other value-based organizational theories to understand FLOSS participation. For example, resource-based theory could be leveraged to understand how certain community-based resources could cause new contributor participation or even how FLOSS communities could sustain a competitive advantage, however that concept might be conceptualized, in the FLOSS marketplace.

A potential limitation to our work is associated with the tools that we used to collect our empirical data. Our tools allowed us to collect data only about completed transactions; that is, we were not able to collect information pertaining to developers' attempts to submit patches that were rejected. This information resides in the project trackers and developer mailing lists related to individual projects. Given that each project uses a different system, the complexity of the tools that we would have needed to collect all of this

data would have been an order of a magnitude more complex than what we have developed and used for this study. This information, however, could add further insight into the effects of contribution costs and community structures, and future studies could benefit from collecting and making inferences from such information. This was more of a trade-off than a limitation, though, as we chose generalizability over rich data. To our knowledge, this is the only FLOSS-related work that attempts to base large-sample observations on an established theory like TCT.

Furthermore, we have alluded to the possible influence of having star programmers join the development effort of a FLOSS community and how that might increase the popularity of a project and result in greater numbers of new contributors. While ideally, we would have liked to capture such information, the scope of the projects and the timelines that we have examined made it impossible for us to know which programmers are considered stars. We attempted to make up for this limitation by controlling for the effect of popularity as reported by FLOSS developers on ohloh.net. If it was at all possible to identify the time periods in which star programmers joined a FLOSS projects, future studies could provide a better understanding of the impact of such an effect.

Some of the empirical findings were also a surprise and warrant further investigation to understand more fully the nature of the relationships they represent. For example, while we have not predicted the non-linear nature of the relationship of knowledge codifiability, amount, and relatedness, we offered post-hoc analysis that could establish the basis for future investigations. Follow-up studies could examine the quadratic relationships and treat them explicitly using deductive theory. While we now know that there is a

certain threshold at which benefits associated with knowledge codifiability and knowledge diversity turn into liabilities, it is difficult to establish a priori what these threshold points are. This does not however diminish the value of this finding, as FLOSS communities are now made more informed of the complexities associated with managing community sustainability.

It is also important to note that this is the first attempt that we know of that uses the leading eigen-vector method of community structure identification (Newman and Girvan, 2004; Newman, 2006b) to assess the dependency structure and modularity of a software code base. Given that the modularity-based variable observed both discriminant validity and a significant relationship to the dependent variable in the direction predicted by our theory, this leads us to conclude that there is validity to the measure. However, given that this is a new measure, we have yet to establish meaning to subtle variations in the measure to be able to use it in more practical settings. Although the measure serves well in comparing which code base is more modular relative to another code base and assessing the design improvements for the same code base by comparing new values to previous values of modularity, we still do not have a complete understanding of what the implications are for absolute modularity values in and of themselves. We believe that further use of this method could provide a better understanding of these values and their implications and, more importantly, an understanding of the strengths and weaknesses of this method. For example, further use of this method would allow us to determine whether a modularity value of .9 has indeed significant management implications when compared to a code base with a modularity of .5 or .89. In addition, further investigation is needed to under-

stand the nomological network in which such a measure is embedded in order to understand whether the measure can serve project managers well as a standalone measure or whether it should be considered relative to other metrics, such as code size, team size, and number of modules.

7 Conclusion

In this study, we conceptualize FLOSS communities as competitors in both the software and FLOSS marketplaces. FLOSS communities compete in the software marketplace to get more people to use their software products. In the FLOSS marketplace, communities compete for a larger portion of contributor mindshare, contributors receive community-related benefits, and the community benefits from advancing its software development effort. The two marketplaces are related in that development efforts in the FLOSS marketplace would allow the community to become more competitive in the software marketplace by improving its software offerings. At the same time, FLOSS communities compete on converting users of their software to contributors that participate in the FLOSS marketplace. We take particular interest in the conversion of users into first-time contributors and use the insights from TCT to explain how contribution transaction costs are the main impediment to initial contributions. We identify the characteristics of knowledge, codifiability, completeness, and diversity as the main sources of contribution-related transaction costs. We then provide empirical support for our theory.

We contribute to the body of knowledge on FLOSS community management by finding empirical support to the idea that the codification of requisite development knowledge

by means of documenting the source code is important in reducing the effort required of potential contributors. By reducing the costs associated with acquiring requisite knowledge, a FLOSS community can increase the number of developers who make a contribution to its source code. However, such costs can only be managed up to a point. After that point navigating through the code base starts to become cumbersome and might deter developers from contributors.

In line with the theory forwarded by Baldwin and Clark (2006), we contribute by finding empirical support for the idea that diverse and complete development knowledge, which is reflected by modularity, size, and the number of modules in a code base, is related to the number of contributors. These findings highlight the importance of properly designing software to encourage participation. Modular designs that minimize dependencies across modules and maximize them within modules are designs that encourage participation. Maximizing dependencies within a module requires developers to group highly related modularity within a module, making it easy to find by potential contributors. Reducing the dependencies across modules would help make changes in one module have little effect on other modules, thereby enabling parallel development and more predictable software that is easier to manage (Baldwin and Clark, 2000; Page-Jones, 1998).

The relationship of the size and number of modules to contribution in particular might suggest that there are certain types of software products that have a tendency to attract contributors if the software was developed by a FLOSS community. We attributed this to the reduction in risk associated with the development effort since contributors get to benefit from the software and pool their efforts, thereby distributing the risk of failure

over a larger group of contributors. This situation would be a more rational option than developing such software alone. However, we also found that having a very large number of modules might also come at a cost and deter participation since developers will have a hard time finding their way around the code base in order to make their changes.

The final contribution we make is the utilization of the leading eigen-vector method for community structure identification (Newman and Girvan, 2004; Newman, 2006a) and the graph modularity measure (Newman, 2006b) as software metric tools. We also contribute by finding evidence of the validity of the modularity measure. This evidence is observed through the support of the relationship between modularity and sustainability that was predicted prior to obtaining the result in addition to the discriminant validity of the measure.

Our contributions have important implications for both theory and practice. As part of the theoretical implications, we clarify the theoretical distinction between the FLOSS marketplace and the software marketplace. This distinction will have profound theoretical impacts since it differentiates between the dynamics of usage of FLOSS through the software marketplace and those of development through the FLOSS marketplace. Furthermore, our conceptualization of the marketplace contributes to the discourse on whether FLOSS development represents a cathedral or a bazaar (e.g. Bezroukov, 1999; Raymond, 2001; Krishnamurthy, 2002; Crowston and Howison, 2005). What our framing suggests is that the constellation of all the FLOSS communities can be thought of as a babbling bazaar in which users and developers can pick and choose the software they use and the projects they work on. After all, a bazaar is merely a marketplace.

As with any marketplace, competition is a very important aspect that is not yet understood well in the FLOSS context. Our framing also allows us to highlight some competitive forces in the FLOSS marketplace and to use established theories in organizational literature, like TCT, to gain a better understanding of the dynamics in the FLOSS marketplace. This opens up the possibility of using other organizational theories that focus on competition to understand the competitive nature of FLOSS communities more fully.

In addition, we outline the importance of understanding the structure of the source code as a proxy for requisite knowledge given that it is an explicit form of that knowledge. We also propose the use of new methods that offer rigorous and objective means to compare and examine software structures (i.e., source code dependency graphs). Since these tools measure aspects that are important in software engineering efforts, such as modularity, they can be used to assess the impact of initiatives that aim to improve a software's structure, whether it be in FLOSS communities or software engineering projects.

Overall, the results support the basic idea behind our theory, namely that FLOSS contributors are rational value-maximizing actors that will refrain from contribution when the cost to contribute is higher than the returns gained from participation. In addition, the results also lend support to the existence of transaction costs in the context of FLOSS participation and are closely tied to the characteristics of the code base. Both ideas are supported by our empirical findings that support our hypotheses and the diminishing returns of knowledge codifiability, amount, and relatedness. This suggests that managing the costs of participation is indeed an effective strategy to encourage new contributions

and maintaining the sustainability of FLOSS communities. The interplay between cost to participate and the value of that participation is yet to be understood and is a good direction for future research. Such research could provide a better explanation to the observation of diminishing returns from knowledge codifiability, amount, and relatedness. In addition, our empirical findings suggest that the knowledge diversity construct is multidimensional as suggested by Turner and Makhija (2006) given the distinct effect for each.

Another important practical implication of our findings is our outline of the importance of activities that are usually considered to be chores in FLOSS communities, such as code reorganization and documentation. The importance of these activities not only lies in how they improve the maintainability of a code base but also in their ability to increase the number of new contributors.

Appendices

Appendix A: Modularity Related Constructs

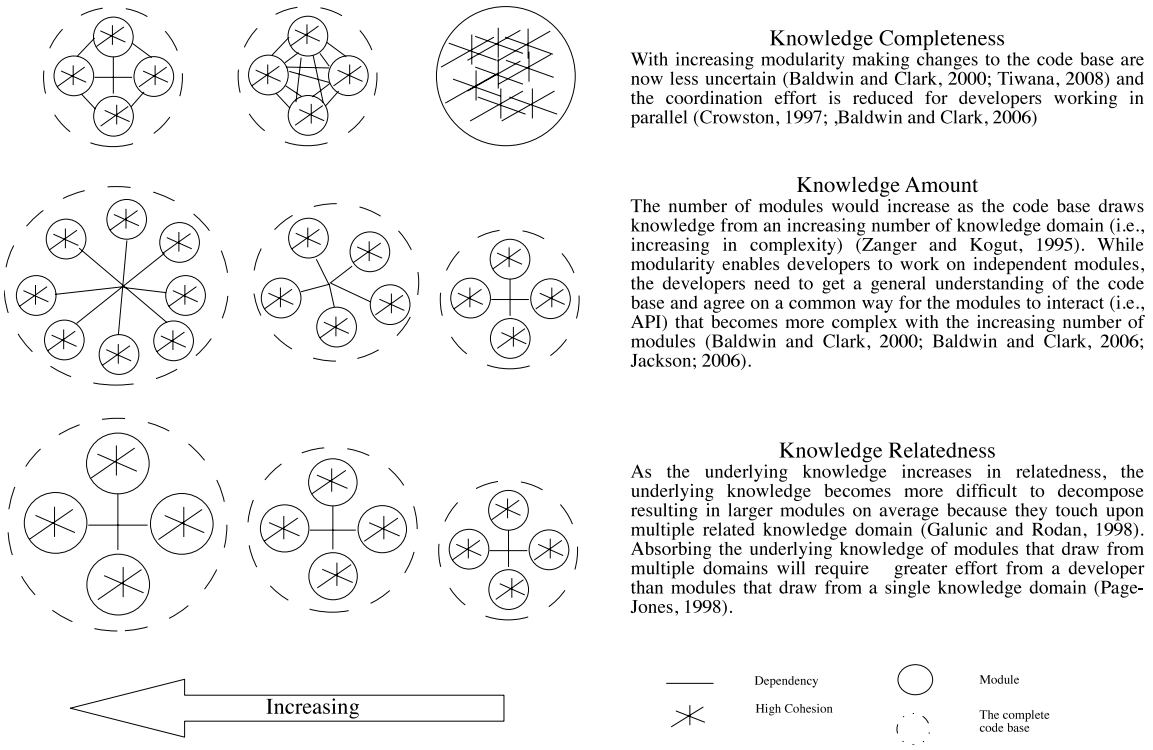

Figure 2.7: How the different modularity-related constructs vary independently

Appendix B: Analyzing Contributors

The following message is an example of a log message from the Django project revision control system that indicates a patch was made by a contributor with the pseudo-name nfg:

> Fixed #9859 Added another missing force_unicode
>
> needed in admin when running on Python 2.3. Many
>
> thanks for report & patch to nfg.

- (Django Revision 9656)

We wrote automation scripts to identify the names of contributors from log messages by looking for certain keywords in the log messages such as "thanks" or "patch." While identifying the names from log messages might be a trivial task when performed by humans, automating such a task was quite involved given that projects identified contributors differently and that the same individual could use multiple spelling for his/her name. Nevertheless, we undertook this task and developed the automated scripts. Once we completed the scripts, we needed a way to assess their reliability. Since we knew that the same process could be performed reliably by a human, we assessed the inter-rater reliability for the results of a manual process relative to the results of an automated process for the name identification. Towards that end, we first randomly sampled around 100 log messages from each of the projects in our sample. We then counted the names identified from each log using both a manual process (i.e., human) and an automated process that

used the automation scripts we developed. We then computed the inter-class correlation (ICC) between the results produced by the manual relative to the results produced by the automated processes. Using the ICC measure as an estimate of the inter-rater reliability (Shrout and Fleiss, 1979), we obtained a value of 0.921 for a total of 18500 observations from 185 projects from our sample. This result led us to conclude that the results from the automation scripts and manual process were interchangeable and that we could proceed to use our automation scripts to identify the names of contributors reliably.

Notice, however, that the automation scripts were used only on 185 projects from the total 235 projects in our sample. The reason we could not use the automation scripts on some of the projects is that these projects identified a ticket number in the commit log messages rather than attributing individuals. For such projects, we wrote a different set of automation scripts that collected these ticket numbers and used them to retrieve contributor information from the project tracking website[16]. Only information related to contributions made between the beginning of January 2007 and the end of June-2009 was collected.

After obtaining the names of the authors from all of the projects in our sample, we then proceeded to clean the data to reduce any errors associated with the automated collection of names. We first manually cleaned the names by removing any references that did not represent real authors. For example, when terms like "bug #1234" were captured as authors, they were discarded. We only retained names, pseudo-names, or emails.

---

[16] The project tracking website is the website used to list known bugs and list feature requests. Some FLOSS projects rely more heavily on such websites for their development process, where contributors would post their patch for review on the project tracker before it is reviewed and committed.

Following the cleaning step, we proceeded to normalize the names to ensure that different reference to the same individual counted as a single author rather than different authors. We developed an application that listed all the identified names of all committers and contributors within a project and then made a all the other names that could potentially be variations of identified names. For example, a name like "John Doe" would have all of the following names listed as potentially related to it should they exist in the system: "John," "Doe,""john.doe@gmail.com," and "jdoe". We then manually confirmed which names were related, and the system then counted all of these related names as a single individual, whether for a committer or contributor. In cases in which the authors were identified as "Guest," "anon," or "anonymous" because they forgot to mention their names, we normalized the names following the recommendation of Howison and Crowston (2004) to count only a single anonymous contributor as unique per single commit.

We then proceeded to count the number of new contributors within an analysis period. To identify new contributors, we counted only individuals who had not been referenced in prior analysis periods either as contributors or committers. Since we had no history about contributors prior to 2007, we excluded the first three months of our data from the analysis and used it only as a means of identifying new contributors for subsequent periods. It is important to note that the normalized names were used in this counting process to avoid inflating the numbers of new contributors when a individuals used different spellings of their names to identify themselves.

Appendix C: Measuring Modularity

To measure modularity, we utilized the leading eigen-vector method of community structure identification (Newman, 2006a) to partition the dependency graph of the source code. This partitioning method is similar in nature to principle component analysis (PCA) (Newman, 2006a). While PCA uses a covariance matrix to cluster items and maximize the explained variance in the data, the leading eigen-vector method uses the eigen-vectors of the modularity matrix to divide a graph continually while maintaining the maximum possible modularity measure (Newman and Girvan, 2004) in order to discover groups.

Modularity is a measure of how cleanly a graph is partitioned (Newman and Girvan, 2004; Newman, 2006b). It takes into account the groups identified by any clustering method, such as the leading eigen-vector method, and then gives a probability of how random the edges are distributed in a graph. A great number of edges within groups and a low number of edges between vertices belonging to different groups indicate that the distribution of edges follows a pattern and is not random and that the modularity value will be closer to one. When the edges are random, the probability that an edge goes from one node to another will be the same. In such graphs, we will not see any particular edge pattern for any group. When the graph is partitioned, the number of edges leaving a group will not differ from the number of edges within a group, which results in an inefficient partitioning of the graph with modularity values close to or possibly lower than zero[17].

[17] According to Newman (2006a), modularity values that are less than zero are equivalent to zero.

So, the question that the value of modularity would answer would be "are we seeing more than the expected number of edges in a group, and less than expected number of edges spanning groups?"

When the software code dependency graph is partitioned using the leading eigenvector method, the files are classified into modules to maximize the edge count (i.e., dependencies) between files that are in the same module, while minimizing the count of edges that span modules. Just like PCA, items that hold together well (i.e., exhibit cohesion) are identified as modules. However, a lack of edges between modules, as in PCA, would signify how distinct each module is (i.e., exhibits loose coupling). Once the structure was identified, we extracted the modularity measures of the partitioned source code dependency graph and used it as an estimate of the modularity of the code base design.

References

Ackoff, R. (1967). Management misinformation systems. *Management Science*, 14(4): B147-B156.

Aguinis, H., Beaty, J. C., Boik, R. J., & Pierce, C. A. (2005). Effect Size and Power in Assessing Moderating Effects of Categorical Variables Using Multiple Regression: A 30-Year Review. *Journal of Applied Psychology*, 90(1), 94-107.

Aiken, L. S. and West, S. G. (1991). *Multiple Regression: Testing and interpreting interactions*. Sage Publications, Inc.

Alavi, M. and Leidner, D. E. (2001). Review: Knowledge management and knowledge management systems: Conceptual foundations and research issues. *MIS Quarterly*, 25(1):107-136.

AlMarzouq, M., Zheng, L., Rong, G., and Grover, V. (2005). Open source: Concepts, benefits, and challenges. *Communications of AIS*, 2005(16):756-784.

Attewell, P. (1992). Technology diffusion and organizational learning: The case of business computing. *Organization Science*, 3(1):1–19.

Baldwin, C. Y. and Clark, K. B. (2000). *Design rules, Vol. 1: The power of modularity*. The MIT Press.

Baldwin, C. Y. and Clark, K. B. (2006). The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116-1127.

Bates, D. and Maechler, M. (2009). *lme4: Linear mixed-effects models using S4 classes*. R package version 0.999375-31.

Bezroukov, N. (1999). Open source software development as a special type of academic research: Critique of vulgar raymondism. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/696/606.

Bonaccorsi, A. and Rossi, C. (2005). Altruistic individuals, selfish firms? http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1476.

Brooks, F. (1975). The mythical man-month. In *Proceedings of the international conference on reliable software*, volume 10. ACM Press.

Capiluppi, A. and Adams, P. J. (2009). Reassessing brooks' law for the free software community. In Boldyreff, C., Crowston, K., Lundell, B., and Wasserman, A. I., editors, *OSS*, volume 299 of *IFIP*, 274-283. Springer.

Champoux, J. E., & Peters, W. S. (1987). Form, effect size and power in moderated regression analysis. *Journal of Occupational Psychology*, 60(3), 243-255.

Choudhury, V. and Sampler, J. L. (1997). Information specificity and environmental scanning: An economic perspective. *MIS Quarterly*, 21(1):25-53.

Clemons, E. K. and Hitt, L. M. (2004). Poaching and the misappropriation of information: Transaction risks of information exchange. *Journal of Management Information Systems*, 21(2):87-107.

Clemons, E. K., Reddi, S. P., and Row, M. C. (1993). The impact of information technology on the organization of economic activity: The "move to the middle" hypothesis. *Journal of Management Information Systems*, 10(2):9-35.

Coase, R. H. (1937). The nature of the firm. *Economica*, 4(16):386–405.

Cohen, J., Cohen, P., West, S., and Aiken, L. (2003). *Applied multiple regression/ correlation analysis for the behavioral sciences*. Lawrence Erlbaum, third edition.

Cohen, P., Cohen, J., West, S. G., and Aiken, L. S. (2002). *Applied multiple regression/ correlation analysis for the behavioral sciences, Third Edition*. Lawrence Erlbaum, third edition.

Cohen, W. M. and Levinthal, D. A. (1990). Absorptive capacity: A new perspective on learning and innovation. *Administrative Science Quarterly*, 35(1):p128- 152.

Collins-Sussman, B., Fitzpatrick, B. W., and Pilato, C. M. (2004). *Version Control with Subversion*. O'Reilly Media, Inc., Sebastopol, CA.

Conner, K. R. and Prahalad, C. K. (1996). A resource-based theory of the firm: Knowledge versus opportunism. *Organization Science*, 7(5):477-501.

Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science*, 8(2):157-175.

Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1207.

Dahlander, L. and Magnusson, M. G. (2005). Relationships between open source software companies and communities: Observations from nordic firms. *Research Policy*, 34(4):481-493.

Darcy, D., Kemerer, C., Slaughter, S., & Tomayko, J. (2005). The structural complexity of software an experimental test. Software Engineering, IEEE Transactions on, 31(11), 982-995.

English, R. and Schweik, C. M. (2007). Identifying success and tragedy of floss commons: A preliminary classification of sourceforge.net projects. In *FLOSS '07: Proceedings of the First International Workshop on Emerging Trends in FLOSS Research and Development*, 11, Washington, DC, USA. IEEE Computer Society.

Fichman, R. G. and Kemerer, C. F. (1997). The assimilation of software process innovations: An organizational learning perspective. *Management Science*, 43(10):1345-1363.

Galunic, D. C. and Rodan, S. (1998). Resource recombinations in the firm: knowledge structures and the potential for schumpeterian innovation. *Strategic Management Journal*, 19(12):1193-1201.

Gardner, W., Mulvey, E. P., and Shaw, E. C. (1995). Regression analyses of counts and rates: Poisson, overdispersed poisson, and negative binomial models. *Psychological Bulletin*, 118(3):392 -404.

Goetz, B., Peierls, T., Bloch, J., Bowbeer, J., Holmes, D., and Lea, D. (2006). *Java Concurrency in Practice*. Addison-Wesley Professional.

Grant, R. M. (1996a). Prospering in dynamically-competitive environments: Organizational capability as knowledge integration. *Organization Science*, 7(4):375- 387.

Grant, R. M. (1996b). Toward a knowledge-based theory of the firm. *Strategic Management Journal*, 17:109- 122.

Gulati, R., Lavie, D., & Singh, H. (2009). The nature of partnering experience and the gains from alliances. *Strategic Management Journal*, 30(11), 1213-1233.

Howison, J. and Crowston, K. (2004). The perils and pitfalls of mining sourceforge. In *In Proceedings of the International Workshop on Mining Software Repositories (MSR 2004*, pages 7-11.

Jackson, D. (2006). *Software abstractions: logic, language, and analysis*. The MIT Press.

Koch, S. (2004). Profiling an open source project ecology and its programmers. *Electronic Markets*, 14(2):77-88.

Kogut, B. and Zander, U. (1992). Knowledge of the firm, combinative capabilities, and the replications of technology. *Organization Science*, 3(3):383-397.

Kogut, B. and Zander, U. (1996). What firms do? coordination, identity, and learning. *Organization Science*, 7(5):518, 502.

Krishnamurthy, S. (2002). Cave or community? an empirical examination of 100 mature open source projects. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/960/881.

Krogh, G. V. and Hippel, E. V. (2006). The promise of research on open source software. *Management Science*, 52(7):975-983.

Kuk, G. (2006). Strategic interaction and knowledge sharing in the KDE developer mailing list. *Management Science*, 52(7):1031-1042.

Lakhani, K. R. and Wolf, R. G. (2007). *Why hackers do what they do: Understanding motivation and effort in free/open source software projects*, 3-22. The MIT Press.

Laporte, L. and DiBona, C. (2006). Interview with guido van rossum. Podcast.

Lavie, D. (2007). Alliance portfolios and firm performance: A study of value creation and appropriation in the U.S. software industry. *Strategic Management Journal*, 28(12), 1187-1212.

Lee, G. K. and Cole, R. E. (2003). From a firm-based to a community-based model of knowledge creation: The case of the linux Kernel development. *Organization Science*, 14(6):633-649.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution—the nineties view. In *METRICS '97: Proceedings of the 4th International Symposium on Software Metrics*, page 20, Washington, DC, USA. IEEE Computer Society.

Lerner, J. and Tirole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics*, 50(2):197.

Liu, X. and Iyer, B. (2007). Design architecture, developer networks and performance of open source software projects. In *ICIS 2007 Proceedings*.

MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015-1030.

Malone, T. W. and Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87-119.

Malone, T. W., Yates, J., and Benjamin, R. I. (1987). Electronic markets and electronic hierarchies. *Commun. ACM*, 30(6):484-497.

Mata, F. J., Fuerst, W. L., and Barney, J. B. (1995). Information technology and sustained competitive advantage: A resource-based analysis. *MIS Quarterly*, 19(4):487-505.

McClelland, G. H., & Judd, C. M. (1993). Statistical difficulties of detecting interactions and moderator effects. *Psychological Bulletin*, 114(2), 376-390.

Midha, V. (2008). Does complexity matter? the impact of change in structural complexity on software maintenance and new developers' contributions in open source software. In *ICIS 2008 Proceedings*.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309-346.

Newman, M. E. J. (2006a). Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74:036104.

Newman, M. E. J. (2006b). Modularity and community structure in networks. *PNAS*, 103:8577.

Newman, M. E. J. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, 69:026113.

Nieuwenhuis, R., Pelzer, B., and te Grotenhuis, M. (2009). *Influence.ME: Tools for detecting influential data in mixed effects models*. R package version 0.7.

Oh, W. and Jeon, S. (2007). Membership herding and network stability in the open source community: The ising perspective. *Management Science*, 53(7):1086-1101.

O'Mahony, S. (2003). Guarding the commons: How community managed software projects protect their work. *Research Policy*, 32(7):1179-1198.

Ouchi, W. G. (1980). Markets, bureaucracies, and clans. *Administrative Science Quarterly*, 25(1):129-141.

Page-Jones, M. (1998). Cohesion. In The Practical Guide to Structured Systems Design. Retrieved October 6, 2008, from http://www.waysys.com/ws_content_bl_pgssd_ch06.html

R Development Core Team (2009). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria.

Ramm, M. (2008). DjangoCon 2008 keynote: Mark Ramm.

Raymond, E. (2001). *The cathedral and the bazaar: Musings on linux and open source by an accidental revolutionary*. O'Reilly, Cambridge, MA, revised edition.

Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. *Computer*, 40(4):25-32.

Roberts, J. A., Hann, I., and Slaughter, S. A. (2006). Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984-999.

Sanchez, R. and Mahoney, J. (1996). Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal*, 17:76, 63.

Schweik, C. M., English, R. C., Kitsing, M., and Haire, S. (2008). Brooks' versus Linus' law: an empirical test of open source projects. In *dg.o '08: Proceedings of the 2008 international conference on Digital government research*, pages 423-424. Digital Government Society of North America.

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

Shrout, P. E. and Fleiss, J. L. (1979). Intraclass correlations: Uses in assessing rater reliability. *Psychological Bulletin*, 86(2):420-428.

Simon, H. A. (1955). A behavioral model of rational choice. *The Quarterly Journal of Economics*, 69(1):99-118.

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.

Tiwana, A. (2008). Does interfirm modularity complement ignorance? a field study of software outsourcing alliances. *Strategic Management Journal*, 29(11):1252, 1241.

Turner, K. L. and Makhija, M. V. (2006). The role of organizational controls in managing knowledge. *Academy of Management Review*, 31(1):197-217.

Ullrich, J., Schermelleh-Engel, K., & Böttcher, B. (2008). The moderator effect that wasn't there: Statistical problems in ambivalence research. *Journal of Personality and Social Psychology*, 95(4), 774-794.

Ven, K. and Mannaert, H. (2008). Challenges and strategies in the use of open source software by independent software vendors. *Inf. Softw. Technol.*, 50(9-10):991-1002.

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.

Warton, D. I. (2005). Many zeros does not mean zero inflation: comparing the goodness-of-fit of parametric models to multivariate abundance data. *Environmetrics*, 16(3):275-289.

Weimer, D. and Vining, A. (2004). *Policy Analysis: concepts andp*. Prentice Hall, fourth edition.

West, J. (2007). Seeking open infrastructure: Contrasting open standards, open source and open innovation. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1913/1795.

Williamson, O. E. (1975). *Markets and hierarchies: Analysis and antitrust implications*. The Free Press, New York, NY.

Zander, U. and Kogut, B. (1995). Knowledge and the speed of the transfer and imitation of organizational capabilities: An empirical test. *Organization Science*, 6(1):p76-92.

Chapter 3

Study 2: Towards a Theory on the Technical Performance of FLOSS Communities

Abstract

Committers in Free/Libre and Open Source Software (FLOSS) communities play a critical role in the progress of the software development tasks in the community. Not only are they some of the most productive members in terms of development activities, but they also get involved by committing the work of other contributors into the community code base, coordinating the development effort, communicating with the broader community for both technical and development support, and participating in most decision-making processes. As a result of this enormous effort shouldered by committers, it comes as no surprise that the FLOSS research found that most committers leave the community they are part of after serving for only a short time period.

Given the critical role that committers play in the progress of FLOSS communities, we question how committers are able to balance the demands of their own development work and their committal duties towards the community and identify the committers as a potential development bottleneck. As a result, we expect the tradeoffs made by the committers to have implications on the pace of progress made by a FLOSS community. Which is why we start questioning Raymond's claim that Brooks' law is obsolete in the FLOSS development context, as some of the tradeoff decisions made by the committers might result in greater development effort, but not necessarily progress.

Using the lens of Organization Information Processing Theory (OIPT), we explain how the technical performance of a FLOSS community is constrained by the committers' capabilities. Our findings suggest that the way in which committers are organized will have a profound implication on the technical performance of the FLOSS community. Furthermore, we find that there is no superior form of organization (i.e., centralized vs. distributed organization of committers); rather, the needs of the community should determine the best organizational form. We provide empirical support for our proposed model and offer some theoretical and practical implications for our findings. We conclude by discussing the limitation of our work and suggest directions for future research.

1 Introduction

Free/Libre and Open Source Software (FLOSS) is produced by online communities in which members' individual development efforts are coordinated with the efforts of other members and are integrated into a coherent whole (O'Mahony and Ferraro, 2007). Members write source code patches that implement a feature or fix a bug, which they then submit for integration into the community source code base to benefit all members (Raymond, 2001).

The idea that a large group of distributed developers can work together to develop software that rivals, if not exceeds, commercial software (AlMarzouq et al., 2005) goes against the conventional wisdom of software engineering. Specifically, FLOSS development seems to go against the ideas forwarded by Fred Brooks', which are referred to as Brooks' Law given how widely they are accepted by software engineers (Brooks, 1975).

One of the ideas from Brooks's Law that clearly goes against the grain of FLOSS development is that developers are required to exert effort in communicating with other team members to maintain the functional integrity of the developed software. However, as more developers are added, conventional software projects start to experience delays because more of the developers' time is wasted on communication. Based on this law, the development performance of large FLOSS communities, such as the Linux Kernel, could not be possible (e.g., Kroah-Hartman et al., 2008).

In his seminal work *The Cathedral and The Bazaar*, Eric Raymond (2001) drew on his personal experiences in FLOSS development to provide an explanation as to why Brooks' Law does not apply in the case of FLOSS development. Unlike the significant effort required in traditional software development teams (Brooks, 1975), Raymond suggests that communication requirements between FLOSS developers is minimal because the source code can convey much of the information needed for coordination. Such source-code-enabled communication allows for distributed development, which Raymond described as a babbling bazaar. In addition, Raymond highlights the benefits of the FLOSS practice of releasing software updates early and often. These benefits not only include improved software quality due to the early discovery and correction of bugs, but more importantly, developers can contribute incremental changes that reduce the effort needed to communicate with one another and can thus increase the development performance of the community.

However, more recent literature began to find differences between many FLOSS projects and the way Raymond described them, thereby raising questions about the assumptions we hold regarding FLOSS. For example, while Raymond described FLOSS communities as being distributed babbling bazaars, Krishnamurthy (2002) found that the majority of FLOSS projects hosted on sourceForge.net were highly centralized, describing them instead as caves. Mockus et al. (2002) and later Crowston and Howison (2005) found that FLOSS communities varied in how they organized their development activities. More interestingly, it seems that empirical support was found for both Raymond's

(eg. Koch, 2004; Schweik et al., 2008) and Brooks' (Capiluppi and Adams, 2009, eg.) views.

The opposing results from the literature suggest that both views are likely to be reconciled and that there is much that we have yet to know about the development performance of FLOSS communities. Specifically, we would like to understand if one of the views prevails due to the variations found in how FLOSS communities organize development and whether this variation will have any implication on performance. This will be important to both theory and practice given the increased interest in FLOSS by for-profit organizations (Fitzgerald, 2006). Furthermore, the transformation of the FLOSS phenomenon (Fitzgerald, 2006) tells us that it will be important to revisit our understanding of the development process in FLOSS communities and that we need to ground our understanding of its performance in established theory.

To state our research question formally, we want to know: "Is there a relationship between the organization of committal activities and development performance in a FLOSS community, and what factors will it be contingent upon?" To answer this question, we would like to view FLOSS communities in light of traditional organizational theories, which we discuss in more detail in Sec. 3.2.2. Specifically, we leverage insights from Organizational Information Processing Theory (OIPT) (Galbraith, 1973) to reconcile the views of Raymond and Brooks on the performance of FLOSS development teams.

We conclude that no development structure is necessarily better than another; rather, we suggest that FLOSS communities make structural decisions to meet the demands of the tasks that they are performing. Our findings suggest that varying the organization of

development activities might not be the only way to foster performance gains in FLOSS communities. Taking actions to reduce the uncertainty associated with the development tasks, such as code reorganization, might also improve the development performance of a FLOSS community.

To address our research question, this paper will be structured as follows. First, we review the FLOSS development process and identify the critical factors constraining a FLOSS community's performance and identify the committal structure. Then, we review OIPT and explain the notion of fit between the committal structure and tasks in a FLOSS community. Next, we leverage the insights from OIPT to present our research model, which predicts the performance of a FLOSS community based on the fit between the development task and the development structure. In the section that follows, we discuss the methods used to validate our research model empirically. Finally, we discuss the results, implications, and limitations of our study and offer directions for future research.

2 Theoretical Development

2.1 Structures in FLOSS Communities

FLOSS communities have been described as knowledge-sharing and production communities (Lee and Cole, 2003). They integrate the voluntary efforts of individual members into a common pool (i.e., the software), which requires a great deal of effort and coordination (O'Mahony and Ferraro, 2007). The process in which this coordination occurs within FLOSS communities has been described as emergent (O'Mahony and Bechky,

2008), as individuals' roles emerge from the tasks they perform, which they self-select based on their abilities and interests (Raymond, 2001; Lee and Cole, 2003; Bonaccorsi and Rossi, 2003; Crowston et al., 2005; Shah, 2006). Most members participate temporarily and very few continue with the community indefinitely (Shah, 2006; O'Mahony and Ferraro, 2007). Therefore, FLOSS communities' structures tend to be dynamic and evolve along with community needs (Oh and Jeon, 2007).

Although community membership is dynamic, a FLOSS community's structure can be inferred from the patterns of member participation. Crowston and Howison (2005) describe the structure of FLOSS communities as onion shaped, having four different types of community members: the core, the periphery, active users, and passive users (see Fig. 3.1). The development work is conducted by both the core and periphery members. The distinction between these two development groups is that core members perform most of the development work and participate in a more frequent and consistent manner than periphery members (Crowston and Howison, 2005). Bug reports and feature requests could come from members of the core, periphery, or active-user groups. Active and passive users do not contribute to the development but are merely consumers of the FLOSS community product; however, active users differ from passive users in that they contribute feature requests and bug reports to the community.

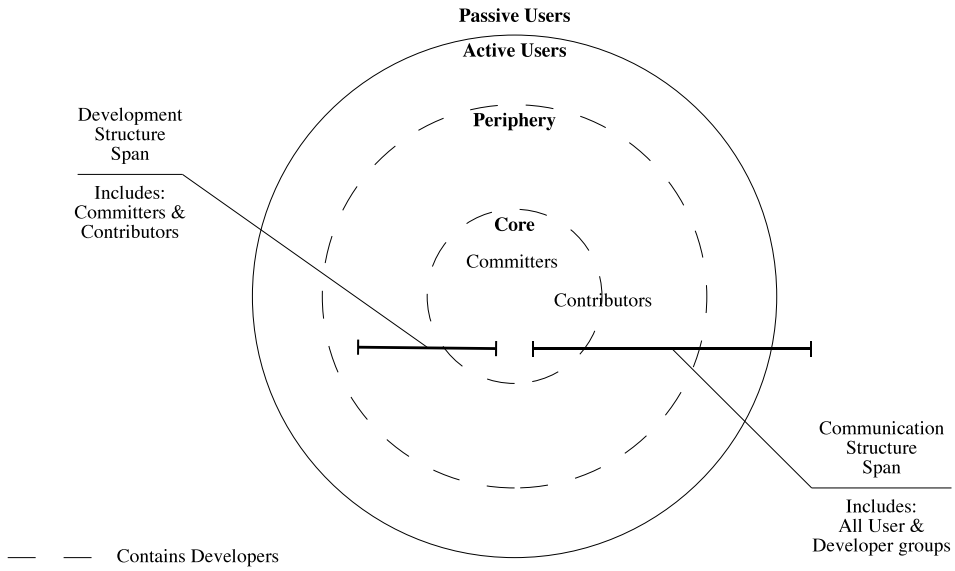

Figure 3.1: FLOSS community structure (adapted from Crowston and Howison, 2005)

There are two distinct substructures in the overall FLOSS community structure: the development structure and the communication structure. The development structure consists of community members who perform development tasks and centers on how the members are organized. The communication structure, on the other hand, spans the whole community and centers on how communication patterns are organized (Mockus et al., 2002; Crowston and Howison, 2005). The development structure is of interest to us since we want to understand the development performance of FLOSS communities.

Within the development structure, there are two different types of members: the committers and the contributors. Both types of members contribute to the software development effort in the FLOSS community, which is why we will refer to them collectively as developers in this work. However, the committers differ in that they have access rights to

the community code base. As a result, committers can incorporate changes they make directly into the community code base, while contributors have to work with a committer to do so. Committers rise from the ranks of the contributors after they have proven their trustworthiness and technical competence from their continued contributions to the community (Shah, 2006; Riehle, 2007).

2.1.1 The FLOSS Development Process

In Fig. 3.2 we summarize the steps developers go through before their source code patches are incorporated into the community code base.

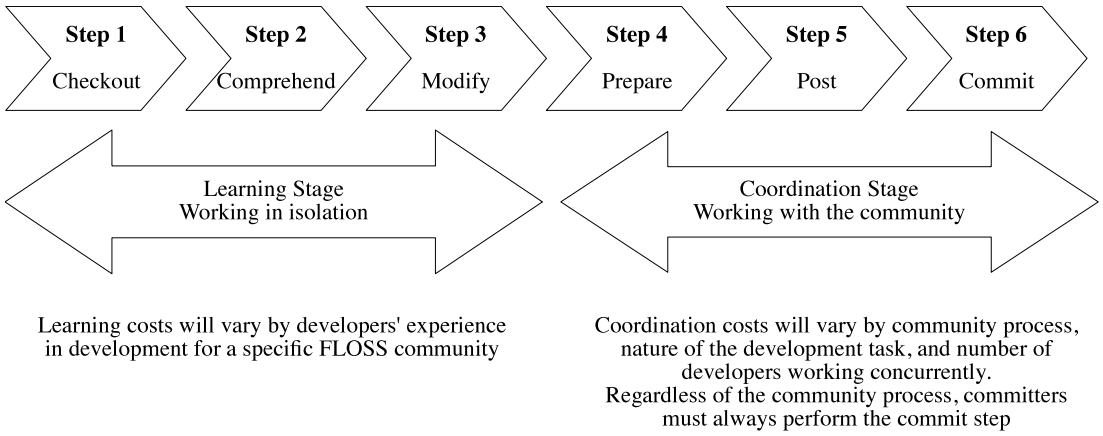

Figure 3.2: Steps required to complete a source code contribution in a FLOSS community

We refer to the first stage a developer goes through as the learning stage. It starts when the developer reviews a fresh copy of the community code base and then attempts to understand its structure and functionality; this is what Brooks refers to as the ramp-up effect

(Brooks, 1975). After acquiring the necessary knowledge to make changes to the code base, the developer then proceeds to make modifications. By the end of this learning stage, the developer has a modified copy of the code base that implements a feature or fixes a bug. Thus far, the developer has worked in isolation and most of the effort he/she has expended is related to learning about the code base.

We refer to the next stage in the contribution process as the coordination stage. The first step of the coordination stage is the preparation step in which the developer prepares a patch that can be contributed to the community. This step includes pulling a recent copy of the community code base and making sure that the recent changes by the contributor do not conflict with changes made my other developers since the initial review step. This step also includes making modifications to the patch to make sure it complies with community coding standards, such as including unit tests or documentation. The developer then posts the patch to the community tracker or mailing list and waits to receive any feedback from the community for possible modifications.

The final step occurs when the patch is committed to the code base by a committer, which may involve further modifications to rectify any problems that the committal step might uncover. At the end of this step the patch is incorporated into the community code base. If the patch was the developer's first contribution to the code base, then the developer is considered a contributor to the community at this time.

The last three steps of the contribution process, which comprise the coordination stage, will require the contributor to work with other committers and possibly contributors in the community. As a result, in this stage any costs are generally related to coordi-

nation and increase with the number of other contributors with whom one needs to coordinate (cf. Brooks, 1975), which is why we refer to it as the coordination stage.

The focus of this work is mainly on the coordination stage. The costs in the coordination stage will vary from one community to another depending on factors that we will identify later in this work. More importantly, however, as we explain in the next section, we expect that committers will be a potential performance bottleneck given the effort required of them before any contribution can be committed.

2.1.2 The Bottleneck

Although committers might initially seem to have an easier job than contributors, they are actually the busiest FLOSS community members, as they are responsible for committing contributions directly to the code base. Not only are they themselves developers in the community, they are also tasked with making decisions about other contributors' work (Shah, 2006), which makes them critical to any contribution. For example, if a contribution is accepted, the committer is tasked with integrating the contributed patch into the code base, which places more responsibility and work on the committer's shoulders, especially when the committed code breaks the work of other developers.

Once we realize the tremendous effort shouldered by committers, it becomes easy to understand why many of them spend only four months on average serving the community after they are promoted (Shah, 2006). Some try to cope with this increased responsibility by limiting their interactions within the community to include only members who they think possess valuable knowledge (Kuk, 2006). To illustrate that this is indeed a problem

that many FLOSS communities face and that it is worthy of community consideration, we present excerpts from the guidelines of some well known FLOSS projects in Tab. 3.1.

Table 3.1: Evidence of delays in FLOSS the development process

Community	Excerpt	Notes
Subversion	If you don't get a response for a while, and don't see the patch applied, it may just mean that people are really busy.	Patch committals can experience delays.
Mozilla	Getting attention: If a reviewer doesn't respond within a week or so of the review request: • Join #developers ...	Because delays in the review process are all too common, the Mozilla community has a process for how to deal with the problem.
Apache	What if my patch gets ignored? Because Apache has only a small number of volunteer developers, and these developers are often very busy, it is possible that your patch will not receive any immediate feedback. Developers must prioritize their time, dealing first with serious bugs and with parts of the code in which they have interest and knowledge. Here are some suggestions on what you can do to encourage action on your patch:...	Delays in patch committal are all too common and the community explains the reasons and gives suggestions on how to alleviate the problem.

Since all development work must involve a committer and committers are the busiest members of the community, we conclude that their activities are the bottleneck to the FLOSS development process (cf. Goldratt and Cox, 1994). This means that the performance of a community will depend on the performance of the committers and that any performance impacting optimization should target committers' (Goldratt and Cox, 1994). As

a result, we conclude that the way in which workload is distributed among the committers will be an important determinant of performance.

2.1.3 The Committal Structure

The committal structure refers to how committal workload is distributed among the committers. The committal structure can be centralized, if committal activity is performed by a focal group that is considered small relative to the size of the development structure's membership. The committal structure can also be decentralized when this focal group represents a larger portion of the developers. A highly centralized committal structure would include only a single committer, whereas, a highly decentralized structure would include all the developers as committers.

Somewhat contrary to the babbling bazaar description by Raymond (2001), which represents a decentralized committal structure, FLOSS communities display both centralized (e.g. Krishnamurthy, 2002; Crowston and Howison, 2005) and decentralized tendencies (Mockus et al., 2002; Crowston and Howison, 2005). However, the implications of these structures are yet to be understood fully.

To the extent that we can view FLOSS communities as organizations, we can leverage insights from established organizational literature to give us a better understanding of the implications of these structures. The Organizational Information Processing Theory (OIPT) stream of literature (e.g. Galbraith, 1973; Tushman and Nadler, 1978; Tushman, 1979) seems to be the most relevant for understanding committal structures and for reaching the overall goals of this study, as it is concerned with the fit between organiza-

tional structures and organizational performance. Therefore, we will first review the OIPT literature and then revisit committal structures to understand their implications on performance.

2.2 FLOSS and Organizational Information Processing Theory

Organizational Information Processing Theory (OIPT) (Galbraith, 1973) introduces the idea that organizations are information-processing systems that deal with uncertainty in their environment. The theory suggests that organizations that find a fit between their information-processing capabilities and their information-processing needs will perform better than organizations that cannot find such a fit. The theory also implies that there is no single organizational structure associated with superior performance; rather, it suggests that different organizational structures are suited for different conditions (Lawrence and Lorsch, 1967). These insights are important for understanding the implications of the different committal structures in FLOSS communities and their relationships to development performance. Specifically, the theory suggests that neither a centralized nor a decentralized committal structure is better; rather, it suggests that the FLOSS community's needs should determine which structure is a better fit for what developers and users want to accomplish.

Before using OIPT, we first need to demonstrate the theory's relevance in FLOSS communities. Towards that goal, we first need to establish that FLOSS communities can

be viewed as organizations; then we need to establish that the main task performed by such communities involves information processing.

Ahuja and Carley (1999) define a virtual organization as "a geographically distributed organization whose members are bound by a long-term common interest or goal, and who communicate and coordinate their work through information technology." (p. 742). FLOSS communities are a special type of a virtual organization since community members work towards the common goal of software development, are geographically dispersed, and coordinate their work through information technology. So, the first requirement for using OIPT in this context is fulfilled, as FLOSS communities are essentially virtual organizations.

Secondly, software development tasks, which are the main tasks performed by FLOSS communities, have been described as knowledge-intensive (O'Mahony and Ferraro, 2007; Lee and Cole, 2003). Completing these knowledge-intensive tasks requires developers to acquire specific knowledge related to a development process in order to participate effectively (Fichman and Kemerer, 1997; Lee and Cole, 2003). These tasks also require team members to constantly coordinate with one another to ensure that the changes they make to the code base do not conflict with those made by other team members (Crowston, 1997). As depicted in Fig. 3.2, both learning (stages 1 to 3) and coordination (stages 4 to 6) activities are part of the development process in FLOSS communities. Given the specific nature of the knowledge related to development in FLOSS communities (see Study 1), learning or sharing (i.e., coordinating) that knowledge will require extensive interaction and communication among team members (Choudhury and Sampler,

1997; Brooks, 1975), which are considered information-processing tasks (Galbraith, 1973; Crowston, 1997). As such, the second requirement for using OIPT in this context is fulfilled, making OIPT an appropriate lens to understand development performance in FLOSS communities.

2.2.1 The Relationship Between Structure and Performance: Fit

Galbraith (1973) explains the relationship between performance and organizational structure by describing a situation in which subordinate workers are tasked with performing simple tasks. When an unusual circumstance occurs due to a change in the organization's environment, the worker will seek a superior to resolve the situation and will then complete the task.

Such a situation could occur in a FLOSS community in any of the contribution stages depicted in Fig. 3.2. When contributors lack certain knowledge that is required to move through the contribution stages, such as identifying which files need to be changed in order to implement a feature or resolve a bug, they will seek out that knowledge. The committers, having experience with the source code and the community in general, will be the members most likely to possess this knowledge or could at least identify individuals who do. Therefore, contributors constantly seek out committers to assist them in resolving a myriad of issues related to the contribution process.

Before a contribution is committed into the community code base, committers are also required to intervene and, possibly, modify contributed patches before they are committed. As one committer puts it: "I've written before on mailing lists that only about two out

of every five submitted patches I review go in unchanged on a good day and that seems to match other maintainers' experiences, too," [1] suggesting that contributions demand some effort from committers before they are committed. Although the relationship between committers and contributors is not one between a superior and a subordinate, completing a development task will require them to interact in a manner similar to how OIPT describes the relationship between a superior and a subordinate.

Based on OIPT, an organization's structure starts to affect performance when superiors are no longer able to respond to the subordinates' requests for intervention in a timely manner. This situation occurs mainly when the number of requests exceeds the superiors' capacity for handling them, which is a function of the uncertainty inherent in the task performed by the organization (Galbraith, 1973; Tushman, 1979). As a result, tasks are not completed as quickly when superiors are overloaded, and the overall performance of the organization suffers. Thus, such situations occur when the organization's information-processing needs exceed its information-processing capacity, which we refer to as *lack of fit* for short.

It is not surprising that lack of fit would occur in FLOSS communities since committers are mostly volunteers and will very likely have less time to dedicate to community coordination than would a full-time supervisor in a more conventional organization. Furthermore, whatever time committers can afford to give to the community will not be entirely dedicated to resolving contributor issues, as the committers will also be engaged in

---

[1] From the blog of a committer in the Django web framework FLOSS project: http://www.pointy-stick.com/blog/2007/11/02/development-experiences-version-control/

both their own development tasks and interactions with the broader community (Shah, 2006; Crowston et al., 2005; Mockus et al., 2002).

Based on OIPT, FLOSS communities, like organizations, can choose from two different strategies to deal with lack of fit and improve development performance. First, a FLOSS community could increase the information-processing capacity of the committal structure to be able to process more contributions and give committers more time for their own development activities, thereby improving the development performance of the community. Second, the community could take action to reduce their information processing needs by reducing the uncertainty inherent in development tasks. We will refer to both of these strategies collectively as *fit strategies*.

2.2.2 Theoretical Assumptions

The extent to which we expect the fit strategies to work rests on a number of assumptions. The first assumption we make is that development tasks cannot be completed if there are outstanding coordination tasks, which is a reasonable assumption to make in software development as developers need to coordinate to ensure the functional integrity of the software (Brooks, 1975).

The second assumption is about the nature of the coordination task, namely that it involves communication among a group of developers and, therefore, cannot be subdivided. As a result, we cannot have a single individual performing all of the coordination tasks; rather, coordination will demand the involvement of a group of developers, similar to how Brooks (1975) described communication channels between developers.

The third assumption is that the committers are boundedly rational, meaning that they have a limited capacity to process and comprehend information (Simon, 1955). This assumption results in the committal structure having a limited information processing capacity.

The fourth assumption is that there is project-specific knowledge, which developers must gain before performing any development tasks within a FLOSS community, and that this knowledge is immobile in nature and requires extensive effort to transfer and communicate (Choudhury and Sampler, 1997; Fichman and Kemerer, 1997). In addition, this knowledge is heterogeneously distributed among community members. As a result, no two members possess similar knowledge, which makes it difficult to replace and/or add developers without incurring costs related to transferring the requisite knowledge (Brooks, 1975).

Another result of this last assumption is that before new participants can partake in development efforts, they need to obtain the requisite knowledge by either reading the source code or by interacting with knowledgeable community members (Brooks, 1975), which places a burden on these knowledgeable members, including committers. Since committers are also boundedly rational, they will have to make a tradeoff between performing their own development work and assisting other members. This tradeoff could potentially limit the flow of knowledge within the community (Kuk, 2006) and the number of individuals who can effectively participate in the development process; this, in turn, affects the overall performance of the community.

Finally, because committers are the busiest community members and because development progress depends on their intervention, we make a final assumption that the development bottleneck resides within the committal structure and that improving the information-processing capacity of that structure will improve a FLOSS community's development performance (Goldratt and Cox, 1994). Therefore, to expand its capacity for information processing, a committal structure has to be more decentralized, which can be done through the promotion of more committers from the ranks of the contributors. However, based on the assumptions we made, such a strategy will not come without a cost (Tushman and Nadler, 1978), the nature of which we have yet to fully understand. To that end, we formalize our research model in the following section.

2.3 Theoretical Model

Thus far, we have conceptualized FLOSS communities as information-processing systems whose goal is the production of software. Our basic argument, based on OIPT, is that high performing FLOSS communities will be those that manage to find a fit between their information-processing capabilities and their information-processing needs.

We identified committers as being the potential performance bottleneck in FLOSS communities given that they are required to be involved with all contributions and are the busiest members of the community. Therefore, how committers are organized in the community, which we refer to as *committal structure*, will be the main determinant of a FLOSS community's processing capacity (cf. Galbraith, 1973; Tushman, 1979).

As for the information-processing needs of a FLOSS community, OIPT suggests that these needs will emerge from the inherent uncertainty of software development tasks and the environment (Galbraith, 1973). There are, however, many facets to the development tasks that can be sources of uncertainty, which OIPT does not specify. In order to identify the specific sources of uncertainty, we follow the work of Tushman (1979).

Tushman (1979) identified three main source of task-related uncertainty in organizational settings: 1) *task routineness*, defined as the unpredictability of the task, 2) *task environment uncertainty*, defined as the rate of change in the external environment that is beyond the organization's control, and 3) *task interdependence*, defined as the extent to which tasks require coordination to be accomplished.

The same sources of uncertainty exist in FLOSS communities, which we identify in Tab. 3.2. However, the broad nature of the task environment uncertainty construct (Tushman, 1979) in addition to the differences in the competitive dynamics of FLOSS communities and that of the conventional market competitors (see Study 1) requires that we adapt the construct specifically for our context. As a result, we identify first-time contributors as a source of uncontrollable, external environment uncertainty and rename this construct *contributor uncertainty*.

Following OIPT, we forward the model depicted in Fig. 3.3. We identified the centralization of the committal structure as the source of a FLOSS community's information-processing capacity, while task-related sources of uncertainty—task routineness, contributor uncertainty, and task interdependence—are identified as the source of

information-processing capacity. In the following sections, we discuss the logic behind our model in more detail.

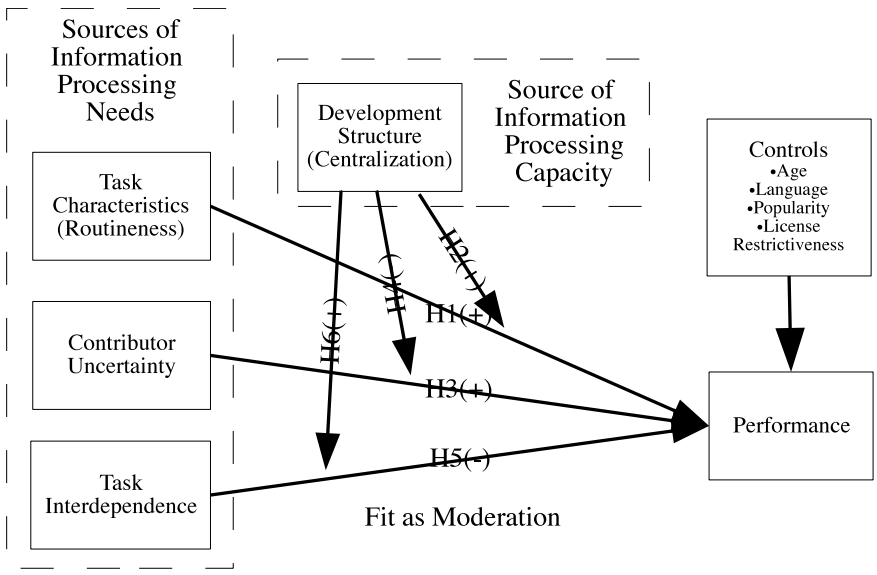

Figure 3.3: Overview of research model

Table 3.2: Overview of theoretical constructs

Construct	Definition
Performance	Progress made towards meeting the demands of the FLOSS community.
Centralization	The degree to which the committal activity in a FLOSS community is concentrated in the hands of a small group of committers relative to the overall size of the development structure's membership
Task Routineness	The degree of predictability in the software-development task done within the FLOSS community.
Contributor Uncertainty	The unpredictability in development tasks that is introduced from the need to integrate the work from contributors who are considered external to the FLOSS development structure.
Task Interdependence	The degree to which developers' development tasks in a FLOSS community require cooperation.

2.3.1 Performance

According to Brooks (1975), effort and progress are two separate concepts. More effort can be expended by a software development project, but that does not guarantee that progress will be made. As Brooks (1975) explains, this occurs when more developers, whose tasks are sequential in nature, are introduced late into a project. The effort needed to bring new developers up to speed (i.e., the ramp-up effect) and coordinate development with them will only detract the most productive developers from making progress, thereby delaying the project. Our concept of performance is similar to Brooks' (1975) notion of progress. As such, we define performance as the progress made towards meeting the demands of the FLOSS community.

The situation with FLOSS development is similar to how Brooks (1975) described progress in traditional development teams. As explained in Sec. 3.2.2, we identified learning and coordination as two important stages in the contribution process (see Fig. 3.2). Committers might become involved in assisting contributors in the learning stage by answering their technical questions and giving them guidance (von Krogh et al., 2003). The effort expended in this learning stage is equivalent to the ramp-up effect described by Brooks (1975). Progress would suffer as the productive committer chooses to trade-off some of his/her development and committal time towards helping other contributors.

The coordination stage is also similar to how Brooks (1975) described software-development teams, as coordination will be required between committers and contributors before a contributed patch can be committed. The number of individuals with whom the contributor or committer need to coordinate may vary depending on who could af-

fected if a problem occurs during the patch-committal process. Since we are assuming that development progress cannot happen if a community's coordination requirements are not met, committers must perform the needed coordination tasks, which detracts them from making progress, thereby delaying the committal of a patch.

While the notions of effort and progress exist in FLOSS communities, progress has to be re-conceptualized for FLOSS communities. Unlike conventional software-development teams for which where progress is measured by how close a team has achieved its deadline or met its project requirements (Espinosa et al., 2007; Gemino et al., 2007; Crowston, 1997; Nidumolu, 1995; DeLone and McLean, 1992), FLOSS communities differ in two regards. First, there are no enforceable deadlines within FLOSS communities because participation is voluntary, which makes it difficult to determine effectiveness in terms of meeting deadlines (Raymond, 2001). Second, there are no preset development requirements for FLOSS communities, which is similar to the conventional waterfall-development model of traditional software creation; rather, requirements are set by members on a needs basis with no guarantee that the needed features will be implemented (Scacchi, 2002).

Given that committers make trade-offs similar to the trade-offs made by productive developers in traditional software development (Brooks, 1975), progress can be observed through the committal activity on the community code base (Koch and Schneider, 2002). A well organized FLOSS project will utilize the time of committers better to produce useful code, rather than requiring them to expend their efforts on coordination and learning activities. More importantly, a commit corresponds to a met demand in the community

signaling that progress has been made (Grewal et al., 2006). Furthermore, a commit only occurs after all coordination and learning issues have been resolved and the committer is satisfied with the quality of the contribution (von Krogh et al., 2003).

Therefore, if two projects have a similar number of committers, holding everything else equal, the team that is able to make more progress is the one that is able to commit more patches within the same time period. The committers for the project that commits fewer patches are probably expending more effort towards coordination and learning, which results in committal delays and fewer patches being committed within a specific time period.

To summarize, our definition of performance fits with the OIPT conceptualization of FLOSS communities as information-processing systems whose goal is the production of software. Software is produced as a result of all of the information processing and coordination tasks completed by committers (Crowston, 1997; O'Mahony and Ferraro, 2007). These information-processing tasks must be handled by the committal structure, and the completion of these tasks occurs when a patch is committed. The committal structure will have a limited capacity for information processing that will be expended towards effort (i.e., coordination and learning) or progress (i.e., patch committal). As the effort requirements increase due to increased uncertainty, progress will be compromised. When effort is minimized by controlling uncertainty, progress is maximized.

2.3.2 Centralization

The most prevalent conceptualization of centralization in recent FLOSS literature (e.g., Grewal et al., 2006; Wu et al., 2007; Tan et al., 2007) is based on the variations of the centrality concept from social network analysis (Freeman, 1979). The different conceptualizations of centralization within the social network literature include: degree centrality, betweenness centrality, and closeness centrality (Freeman, 1979). Such conceptualizations reflect traits of individuals that are part of an underlying communication network. Our conceptualization of centralization departs from this network based understanding to a more classical organizational theory understanding of the construct and is mostly concerned with the capacity of the social or organizational structure to process information.

In organizational literature, centralization has been conceptualized as a characteristic of the organization (Hage and Aiken, 1967; Ouchi and Dowling, 1974). It has been associated with the span of control for supervisors in an organization (Hage and Aiken, 1967). Centralized structures limit control in the hands a few individuals in which a larger group of individuals must report to. Because a supervisor expends greater effort to oversee and communicate with a larger group of employees in centralized structures, he/she is described as having a wide span of control that could result in the overwhelming of the supervisor (Blau, 1968). In decentralized structures however, supervisors will be responsible to oversee and communicate with a smaller group of employees. As a result, they are described as having a narrow span of control in which supervisors are able to dedicate more of their time per employee compared to centralized structures (Blau, 1968).

Span of control doesn't necessarily describe a relationship in which supervisors oversee employees, it is also valid of the relationship between supervisor and employee in

which employees initiate communication and describes the time that a supervisor makes available for an employee (Blau 1968). The relationship between committers and contributors in FLOSS communities might not be that of supervisor and subordinate, nevertheless, span of control could be used to describe the nature of communication between the two within a FLOSS community. Span of control describes the amount of time or effort any committer can make available per contributor, and thus, could be thought of as the capacity to deal with the demands of the contributors.

A committer might have to communicate with a large number of contributors which results in a situation that is analogous to having a wide span of control (i.e., centralized structure). The need to communicate and coordinate with all these contributors is likely to overwhelm the committers' capacity for information processing as the number of contributors increase. A narrower span of control (i.e., decentralized structure) on the other hand is less likely to overwhelm any individual committer and spread the workload over a greater number of committers. As a result, when the span of control for committers is wide, the capacity of the committal structure is likely to be at its limits and is said to be centralized. On the other hand, given the same number of committers and a narrow span of control, the committal structure would still have the capacity to process more information because the committers are less overwhelmed. Such structures are described as being decentralized.

Based on our prior discussion, centralization is defined as the degree to which the committal activity in a FLOSS community is concentrated in the hands of a small group of committers relative to the overall size of the development structure's membership.

While adding committers would make a FLOSS community's committal structure seem more decentralized, how decentralized the committal structure is relative to other communities will depend on the proportion of members of the development structure (i.e. contributors) that posses committal rights. Based on our discussion in Sec. 3.2.1, we expect committers to be the main development bottleneck in a FLOSS community because all development activities will require their involvement. Therefore, we expect that the way in which they are organized will have a profound impact on the community's performance.

FLOSS communities can exhibit both centralized and decentralized development structures. Decentralized structures are expected to provide a greater capacity for processing information, since a larger group of members will not only add to the processing capacity of the structure but will distribute more of the workload to leave smaller tasks for individual committers. However, we do not expect that adding committers will be costless (cf. Tushman and Nadler, 1978) since a larger group of committers could increase the need for coordination (Brooks, 1975; Crowston, 1997). Nevertheless, according to OIPT, we expect both types of committal structure will be the best fit for different development conditions.

2.3.3 Task Routineness

We define task routineness as the degree of predictability in a software development task (Perrow, 1967; Tushman, 1979; March and Simon, 1993). Routine tasks are simple in nature and are reflective of the knowledge required to accomplish them (Turner and Ma-

khija, 2006). Simple tasks will draw from a limited domain of knowledge and will require little effort to accomplish (Zander and Kogut, 1995; Grant, 1996a). Non-routine tasks, on the other hand, have a great degree of unpredictability (Perrow, 1967). Such tasks will also require complex underlying knowledge that draws from a diverse set of knowledge domains (Turner and Makhija, 2006). As a result, non-routine tasks will require individuals to put forth greater effort to accomplish them (Zander and Kogut, 1995; Grant, 1996a).

Assuming that the source code reflects a FLOSS community's underlying development knowledge, non-routine tasks will require the modification or addition of a greater number files. The reason behind such modification is the fact that it is good programming practice to encapsulate similar functionality into the same modules or files (Page-Jones, 1998). FLOSS communities are known to follow good programming practices (MacCormack et al., 2006; Raymond, 2001); therefore, it is probable that contributed patches reflect the complexity of the development tasks.

What is important, however, is how these patches impact committal structures. As the number of files modified by a patch increase, the code that needs to be reviewed by a committer also increases. In addition, the likelihood that such a patch will interfere with the work of other developers will increase as more files are impacted, thereby increasing the chance that the committer will be required to coordinate between developers. Furthermore, since the code is not localized to a specific file, when a patch introduces a bug, the committer's debugging task will be much more complex since he/she is required to trace a larger number of files. As a result, we expect that non-route development tasks

will place a greater amount of coordination requirements on committers, thereby placing a heaver information-processing burden on the committal structure than would routine tasks.

Furthermore, non-routine tasks will place greater learning efforts on committers. Contributors who are required to perform these complex tasks will also be required to change a greater portion of the source file in the code base. Before a change is made to any file, the contributor must understand the contents of that file and understand the best way to make the changes. To obtain that understanding, contributors will not only need to read the source code (Raymond, 2001), but they will also need to seek the assistance of committers in helping them understand the impact of their changes (Krogh and Hippel, 2006). As a result, greater information-processing requirements are placed on the committal structure due to learning requirements associated with non-routine tasks. Hence,

H 1 *Task routineness is positively related to a FLOSS community's performance*

Because routine tasks are simple and highly predictable, they will not generate a great amount of uncertainty and will not require much in terms of information processing. Furthermore, since the requisite knowledge for completing such tasks is easily codifiable, it is easier to communicate by nature (Grant, 1996b, 1996a; Kogut and Zander, 1992), making the time spent by any one individual to learn or communicate development knowledge relatively small. Therefore, we expect routine tasks to put little strain on the development structure in terms of development-related information-processing requirements, making a centralized structure, with its limited information-processing capacity, fit with the completion and management of such tasks. Decentralized structures, on the other

hand, will introduce extra overhead from the need to coordinate between committers (cf. Tushman and Nadler, 1978).

Knowledge of non-routine tasks is more varied and likely to be tacit in nature (Kogut and Zander, 1992), thus making it more difficult to share and communicate (Choudhury and Sampler, 1997). Therefore, non-routine tasks will strain centralized development structures and overwhelm the central committers not only because of the effort required to review complex patches but also because of the extra effort needed to communicate the tacit development knowledge in this technology-mediated environment (Daft and Lengel, 1986).

Given that committers are the most productive developers in a FLOSS community (Shah, 2006; Mockus et al., 2002), a distributed committal structure will allow committers to specialize in different aspects of complex code base development (Grant, 1996a). As a result, committers will need to obtain specialized knowledge about the specific parts of the code base that they work on (von Krogh et al., 2003).

With a greater number of specialized committers, the information-processing requirements are distributed over a greater number of committers, resulting in less effort being performed by any one individual (Tushman, 1979). Coordination and learning activities can then be performed in parallel, resulting in less overall time needed to perform them, even if the time required for any individual task increases due to the extra coordination efforts required with distributed structures.

Therefore, we expect the performance gains for FLOSS communities as development tasks become more routine to be higher under a centralized committal structure than un-

der a decentralized committal structure because of the lack of coordination overhead between committers. It can also be said that as tasks become less routine, the performance gains for decentralized committal structures will be higher than for centralized committal structures because the coordination and leaning tasks being performed in parallel. As a result, a distributed committal structure becomes a better fit for non-routine development tasks. Hence,

H 2 *As tasks become more routine, FLOSS communities with centralized committal structures will have higher performance gains than communities with decentralized committal structures.*

2.3.4 Contributor Uncertainty

Contributor uncertainty is defined as the unpredictability in development tasks that is introduced from the need to integrate the work of contributors who are considered external to the FLOSS development structure. Before contributors can submit patches, they need to acquire the requisite knowledge about the code base that will allow them to make a contribution. Since this knowledge is tacit in nature, it will require a great deal of communication with other knowledgeable developers.

Because the committers are the most knowledgeable about the code base, they will spend a great deal of time communicating with contributors, responding to their questions, and explaining the intricacies of the code base (von Krogh et al., 2003), which places a learning-related burden on committers' information-processing capacities. This is true mostly of contributors who are new to the development process in a FLOSS com-

munity, as returning contributors would likely have absorbed the norms and knowledge related to making a contribution to the community without placing much burden on the committers. This effect is similar to Brooks' (1975) concept of the ramp-up effect.

In addition, new contributors are likely to be the least knowledgeable about the code base. As a result, their patches are likely to introduce more bugs or violate some of the programming guidelines or assumptions that must be adhered to, such as variable access rules and coding conventions. Therefore, patches from new contributors will require more scrutiny and, therefore, more effort from the committers, resulting in delays in the committal of the patch. Therefore, we expect contributor uncertainty to have a negative impact on performance. Hence,

H 3 *Contributor uncertainty will be negatively related to a FLOSS community's performance*.

When a problem occurs during the committal of a contributed patch, the committer will have to read through the code base to find the source of the bug or communicate with other developers to find a solution, both of which are information-processing tasks. As the number of such problems increase, a centralized committal structure will become overwhelmed if it does not promote more committers due to its limited information-processing capacity (Tushman, 1979; Ahuja and Carley, 1999). When the committal structure is no longer coping with the community's information-processing load, external contributions are the first to be ignored by committers who will focus on their own development work. FLOSS communities could cope with lack of fit due to high contributor uncertainty by making the committal structure more decentralized. This, in turn, would

development and decision-making tasks free of constraints by any specific individual (Tushman, 1979).

A decentralized committal structure has the development knowledge and decision-making authority more distributed throughout the community. As a result, the committal and development workload is distributed among a greater number of committers resulting in less information-processing work for each individual committer. Furthermore, since the knowledge is distributed, the coordination and learning activities will not come to a halt when a committer decides to take the time to assist a new contributor. For these reasons, we argue that a decentralized structure is a better fit for situations in which contributor uncertainty is high, as it will result in higher performance gains with increased uncertainty when compared to centralized committal structures. Hence,

H 4 *As contributor uncertainty increases, FLOSS communities with centralized committal structures will have lower performance gains than communities with decentralized committal structures.*

2.3.5 Task Interdependence

When task completion requires the cooperation of multiple individuals or organizational units, it is said to exhibit interdependence (Tushman, 1979; Malone and Crowston, 1994). There are two types of interdependence: intra-unit interdependence and inter-unit interdependence (Tushman, 1979). Intra-unit interdependence occurs when individuals within the same organizational unit are required to cooperate to complete a task, while inter-unit

interdependence occurs when different organizational units within an organization are required to cooperate.

Intra-unit cooperation was found to be much easier than inter-unit cooperation because the members of a single organizational unit possess common knowledge and language, which makes it easier for them to cooperate (Tushman, 1979; Grant, 1996a). Cooperation across organizational units is much more difficult since the language and knowledge differs significantly from one unit to another, making it difficult for members of one unit to cooperate easily with those of another unit. As a result, different organizational units collaborate through designated managers or liaisons who have the necessary skills and authority to work with other units (Tushman, 1979).

A similar organizational pattern can be observed in FLOSS communities in which developers organize around functional software units referred to as modules (Crowston et al., 2005). Organizing source code such that similar and related functionality is encapsulated into the same module is considered good programming practice. Such organization allows modules to exhibit a high degree of within-module interdependency, which is known as cohesion (Page-Jones, 1998). Since similar functionality is contained in single modules, the developers working in each module would share similar knowledge and be able to collaborate as if they were in a single organizational unit in which joint problem solving is important (Tushman, 1979).

Since modules' functionality is limited in scope (Page-Jones, 1998), each module is assigned an owner who is responsible for coordinating its development efforts (Crowston et al., 2005; Crowston, 1997). Within-module coordination is centralized, and it is appro-

priate because of the limited coordination and development information-processing requirements generated from a single module. These requirements are lessened because only a small group of developers who share common knowledge related to the modules are involved in the development, which allows them to coordinate more effectively. Therefore, we argue that within-module coordination does not generate a significant strain on the development structure.

Inter-unit interdependence can also be observed within FLOSS communities when developers working on separate modules are required to coordinate. The need for such coordination is caused by the functional dependencies between modules (Crowston, 1997). Developers working on different modules will possess different sets of knowledge, making coordination more difficult than within-module coordination (Tushman, 1979; Grant, 1996a). Unlike intra-unit dependencies for which problems are localized to the module the developers are working on, problems might not be localized to one particular module when inter-unit dependencies exist. Therefore, managing the effects of inter-unit dependencies is more difficult.

To coordinate in these contexts effectively, not only will the developers need to process coordination information, but they will also need to build common knowledge that will allow them to organize their development tasks (Grant, 1996a). This common knowledge consists of understanding the requisite development information of the other group (von Krogh et al., 2003), which negates the benefits of having a distributed development structure because the developers can no longer specialize in their own modules. In addition to performing all of the coordination tasks, committers will also be required to

perform more development-related information-processing tasks, which forces them to make tradeoffs affecting the overall performance of the community. Therefore,

H 5 *Task interdependency will be negatively related to a FLOSS community's performance*.

As mentioned in the previous section, task interdependence in software development occurs between developers when they are working on the same source code files within a module or when the source code files they are working on have functional dependencies that span modules (Malone and Crowston, 1994; Crowston, 1997). Since within-module dependencies do not put a significant strain on the development structure, it follows that the committal structure will not be strained. As a result, cross-module dependencies become a significant source of development-related information-processing requirements.

Adding new committers will not help address this demand because more committers in addition to the existence of high dependencies would increase the amount of cross-module communication channels needed to maintain the functional integrity of the software (cf. Brooks, 1975; Crowston, 1997). As a result, more of the committal structure's information-processing capacity would be expended in meeting coordination demands. The key to achieving fit in such situations becomes a matter of reducing information-processing requirements in general (Galbraith, 1973).

Since task interdependence is created from dependencies in the source code (Brooks, 1975; Crowston, 1997; Tiwana, 2008), we argue that coordination-related information-processing requirements can be reduced through conscious design decisions that reduce cross-modules dependencies. If source code is designed to exhibit few dependencies be-

tween program units (i.e., modules) and high dependencies within a single unit, then this would allow developers to work concurrently on different program units with little need for coordination between them; such a design is said to exhibit high modularity (Sanchez and Mahoney, 1996; Baldwin and Clark, 2006; Tiwana, 2008).

Limiting cross-module dependencies would limit the unanticipated side effects of making changes in one module, which often appear in different modules. This, in turn, reduces the need for coordination and communication because the development tasks performed by developers would be restricted to specific modules and would enable developers working on different modules to work in parallel (Crowston, 1997). Such a limitation is equivalent to the strategy suggested by Galbraith (1973) for creating self-contained tasks. Furthermore, one of the assumptions Brooks (1975) mentions as to what would lead to an increase in the number of communication channels is the serialization (i.e., dependencies) between tasks. Removing the serialization constraint would reduce the need to communicate between developers.

When coordination requirements are reduced, more of the committers' time is freed for development-related information processing. As a result, the whole community benefits from committers' increased responsiveness to communication and knowledge sharing (Tushman, 1979) and from the specialization that results from modularization (Grant, 1996a). These improvements make the community's overall development more efficient and increase its overall development performance. More importantly, modular designs enable developers to work in parallel, and by reducing coordination requirements, it also reduces the coordination-related costs associated with adding more committers (Baldwin

and Clark, 2006). Therefore, we argue that distributed development is a good fit only when a community improves the modular design of its code base, thereby reducing the task interdependence between developers.

Therefore, what we expect to observe is that FLOSS communities with a distributed committal structure that manage to reduce task interdependence through improved modular designs will exhibit higher performance gains than similar communities with a centralized committal structure. We attribute this higher performance to the reduction of coordination-related information-processing requirements that are removed by the improved source code design. Similarly, FLOSS communities with centralized committal structures and highly interdependent development tasks will perform better than similar communities with decentralized committal structures because decentralized structures will introduce more coordination-related information-processing requirements. Therefore,

H 6 *As task interdependency increases, FLOSS communities with centralized committal structures will have higher performance gains than communities with decentralized committal structures*.

3 Methodology

3.1 Sample

To test our theory, we had to select a sample that satisfied the requirements stated in Sec. 3.2.2.2. Specifically, we needed a sample that represents successful FLOSS projects that enjoy active development, as a great proportion of FLOSS projects are known to be dor-

mant (Krishnamurthy, 2002). Only for actively developed projects, in which committal activity is ongoing over time, could we observe variations in performance. Therefore, our sample frame was limited to successful projects that were likely to have ongoing development activity (Crowston et al., 2003).

Prior literature used sourceforge.net to select their samples, citing that it represented a significant proportion of FLOSS projects (e.g. Krishnamurthy, 2002; Crowston and Howison, 2005; Stewart and Gosain, 2006; Stewart et al., 2006). However, there have been recent reports suggesting that the relevance of SourceForge.net as the main hosting service for FLOSS projects has declined (Paul, 2009a, 2009b). As such, we used ohloh.net, which is a website that lists over 275,000 FLOSS projects regardless of where they are hosted, to find a more representative sample of FLOSS projects. Our choice seems to have grounding, as sourceforge.net purchased ohloh.net to increase its relevance in the FLOSS-hosting market (2009hq).

Following the work of Wu et al. (2007), we selected from the top 1000 most successful projects; however, our sample differed in that we used ohloh.net as the selection website. We got further support that selecting projects from ohloh.net was the right choice when we found that only 22% of the top 1000 projects listed on ohloh.net were hosted on sourceforege.net. Therefore, we expect our results to have more external validity than prior literature that focused on sourceforge.net projects.

3.2 Data Collection

For practical reasons, however, we were unable to analyze all of the projects listed in our sample frame. We found that prior literature focused on a specific programming language (e.g. Midha, 2008; MacCormack et al., 2006). They cite the choice of the language on the size and complexity of the resultant product (Midha, 2008; Wyuker; jones; seemidharefs). Since our goal is to respond to the call to increase the external validity empirical FLOSS studies' findings (Koch, 2004), we expanded our selection to include three of the most widely used languages in FLOSS development that represent the primary programming philosophies: C, C++, and Python. We identified 289 potential projects for analysis representing 28.9% of the top 1000 projects.

Following the recommendations of Howison and Crowston (2004) and prior empirical literature on FLOSS dealing with source code repositories (e.g. Wu et al., 2007; Midha, 2008; Liu and Iyer, 2007), we screened the projects and excluded projects that represented meta-projects and projects with missing history or inaccessible repositories. Since there were no established criteria in prior literature as to what an actively developed project was (besides the ranking on sourceforge.net), we had to establish our own criteria because the majority of the projects in our sample were not listed on sourceforge.net and varied significantly in their level of activity. Therefore, we identified an actively developed project as any project that observed at least a single commit per analysis period since the start of our observation[2]. This left us with a usable sample of 237 projects. We

[2] The closes criteria was the one used by Koch and Schneider (2002) where they identify committers as active if they perform at least a single commit during an analysis period.

then downloaded the source code repository for all of the projects in our sample and proceeded with the data collection.

Prior literature has examined FLOSS projects by observing the activity between releases that spanned several months (Midha, 2008), while others observed the activity for fixed time periods ranging from one month to up to a year (e.g., Wu et al., 2007; Liu and Iyer, 2007; Stewart et al., 2006; Koch, 2004). We chose to go with a fixed period of three months, which we believe is long enough to capture any changes in the community's activity in response to changes in the structure.

Tab. 3.3 provides some descriptive statistics of the projects, which shows that the minimum number of commits per quarter was one. Notice how the number of commits is also skewed towards a high number of commits per month with a mean of 289.2 and a median of 116. The same could be seen for the number of committers, contributors, and popularity, which were measured as number of self-reported users in ohloh.net. This leads us to conclude that our sample is indeed representative of actively developed FLOSS projects. Tab. 3.4 lists the variables we collected from this sample and provides a summary of the corresponding construct definitions and operationalization.

Table 3.3: Descriptive statistics for the project sample

	Median	Mean	STD
Age In weeks relative to 1-1-2007	4	5.64	48.7
Popularity	57.5	192.1	447.71
Committers_{Count per quarter}	5	10.84	14.489
Contributors_{count per quarter}	8	16.29	26.786
Commits_{count per quarter}	116	289.2	456.656

Table 3.4: Variable operationalization

Variable	Definition	Operationalization
Performance (PERF)	Progress made towards meeting the demands of the FLOSS community.	The tallied count of commits listed in the repository within a three month window (Koch and Schneider, 2002; Grewal et al., 2006).
Centralization (CENT)	The degree to which the committal activity in a FLOSS community is concentrated in the hands of a small group of committers relative to the overall size of the development structure's membership.	The ratio of committers to total number of developers in the community (Ouchi and Dowling, 1974).
Task Routineness (TROUT)	The degree of predictability in the software development task done within the FLOSS community.	The average number of files changed per commit.
Contributor Uncertainty (CUNC)	The unpredictability in development tasks that is introduced from the need to integrate the work from contributors who are considered external to the FLOSS development structure.	The ratio of new contributors to the total number of contributors during the analysis period.

Task Interdependence (TINT)	The degree to which the development tasks of the developers in a FLOSS community require cooperation to be completed.	The modularity measure (Newman and Girvan, 2004; Newman, 2006b) of the leading eigen-vector partitioning (Newman, 2006a) of the dependency graph for the beginning of the analysis period.

3.3 Variables

3.3.1 Performance

Content Validity

Performance is defined as the progress made towards meeting the demands of the FLOSS community. Since our theory is built on the premise that the committal structure is the bottleneck for development activities in FLOSS communities, we can assess how many tasks are completed by the committers based on the number of completed commits (Koch and Schneider, 2002; Grewal et al., 2006). We make the assumption that a commit addresses at least one community request by either fixing a bug or implementing a feature. Hence, we can compare the relative performance of communities based on how many issues are addressed while controlling for the differences between these projects.

Procedure

To estimate performance, we counted the total number of commits made by a project performed during a single analysis period. The distribution of the count data suffered from over-dispersion with a dispersion parameter of 56, which prevented us from using a

Poisson-based regression. To address this problem, we performed log transformation to normalize the distribution of the data.

3.3.2 Centralization

Content Validity

Centralization is defined as the degree to which the committal activity in a FLOSS community is concentrated in the hands of a small group of committers. In such cases, the majority of committal activity is done by a relatively small number of committers who divvy the distribution of the workload. When the development structure is more decentralized, committal activity will be distributed among a greater number of committers. In decentralized development structures, the workload is more evenly distributed to a larger number of committers.

To estimate centralization, we calculated the ratio of committers to the total number of developers (i.e., contributors and committers) in a FLOSS community. The intuition behind this measure is that the higher the ratio, (i.e., the greater the number of committers to total developers), the greater the distribution of workload, which is the reverse of centralization. Therefore, higher ratios represent more decentralized structures, while lower ratios represent more centralized structures.

This notion of centrality is very close to the concept of span of control (Simon, 1997) that is measured as number of subordinates to supervisors (Ouchi and Dowling; 1974). While the relationship between a committer and contributor is not that of subordinate and supervisor, the ratio between their numbers reflect the density in upward communication

going from contributors to committers (Ouchi and Dowling, 1974). Low communication density (i.e., smaller ratio) suggests centralized structures where committers expend greater effort having to communicate with more contributors. Whereas low communication density (i.e., greater ratio) suggests a more decentralized structure where the communication effort with contributors is distributed among a larger group of committers.

Procedure

To obtain the centralization estimate, we counted the total number of committers and contributors as identified from the revision control system for the whole analysis period. We then obtained the ratio of committers to total number of committers and contributors as an estimate of decentralization of committal structure. We subtracted that ratio from one to make it a ratio that increased with centralization.

The variable, however, observed a non-normal distribution with distributional masses close to both zero and one. To address this distributional problem, we had to transform the variable into a nominal variable using a median split given the distributional characteristics (MacCallum et al., 2002), where zero represents a decentralized committal structure and one represents a centralized committal structure[3].

3.3.3 Task Routineness

Content Validity

[3] We test the robustness of our results using alternative splits to rule out that our results are an artifact of the median split and use the median split results because they are the easiest to present and interpret. See appendix A for details.

We define task routineness as the degree of predictability in the software development task. Predictable tasks are analyzable and, therefore, easy to break down into simple routines and steps (Perrow, 1967). For software development tasks, simple routines and steps can be implemented and grouped into a single source file. Complex development tasks, on the other hand, will draw from different functional domains (Grant, 1996a) and will require the modification or addition of more than a single source file. Changes to more source files entails that the developer have to acquire the knowledge embedded in each modified file. Furthermore, spreading out the changes to a larger number of files increases the risk of introducing a bug or creating a conflict with another developer, as the likelihood of two developers working on the same file increases (Crowston, 1997). Therefore, the average number of files changed per commit will be inversely related to the routineness of the development task.

Procedure

To obtain the estimate of task routineness we first counted the total number of files changed or added over the analysis period. We then divided that number by the number of commits to estimate the average number of files changed per commit. We then log transformed the variable to normalize its distribution and mean center as recommended by Aiken and West (1991) when testing for interaction terms. As mentioned in the previous section, however, routine tasks are associated with a smaller number of changed or added files. As a result, this proxy will be negatively covaried to variables that truly represent task routineness. Since our variable is mean centered, we multiply it by negative

one to reverse the direction of the variance as to make it positively covaried in relation to task routineness.

3.3.4 Contributor Uncertainty

Content Validity

Uncertainty in task environment is defined as the unpredictability in development tasks that is introduced from the need to integrate the work from contributors who are considered external to the FLOSS development structure. The uncertainty comes specifically from new contributors who are likely to make more mistakes either because they know less about the current code base or about the development process. As a result, committers find themselves spending more time reviewing and committing the work of first time contributors (von Krogh et al., 2003). Furthermore, new contributors will require more of the committers' time to help them get past the ramp-up effect (Brooks, 1975). Returning contributors are likely to have absorbed more of the knowledge required to make a contribution to the code base and to have gotten past the ramp-up effect with a prior contribution. Therefore, we make the assumption that returning contributors are considered part of the development team and will not introduce external uncertainty to the committal structure.

Procedure

We counted the number of unique contributors for the analysis period by extracting their names from the committal logs of the revision control system. We estimated contributor

uncertainty using the ratio of new contributors to the total number of committers. However, given the distributional characteristics of the variable where there are masses close to the values of zero and one, we had to perform a median split and dichotomize the variable into low (zero) and high (one) uncertainty (MacCallum et al., 2002).

3.3.5 Task Interdependence

Content Validity

Task interdependence is defined as the degree to which the completion of development tasks in a FLOSS community requires the developer cooperation. The first form of interdependence we discussed in Sec 3.2.3.5 was intra-unit interdependence, which are the dependencies shared by developers working on the same set of files. We also explained in Sec 3.2.3.5 another form of task-interdependence, inter-unit interdependence, which requires coordination and communication between developers working on different modules. The need for communication across modules is generated by dependencies between the different modules on which developers are working. When organizational units maximize intra-unit dependencies and minimize inter-unit dependencies, it would allow the different units to work in parallel by reducing the communication and coordination requirements (Crowston, 1997).

In software development teams, such separation could be achieved by managing the dependencies between the different functional units (i.e., modules) in the software on which the development team is working (Crowston, 1997). By minimizing coupling between modules, a software development team could reduce the need for communication

between groups working on different modules and could enable a parallel development process (Baldwin and Clark, 2000). On the other hand, developers working on the same modules that encapsulate highly related functionality will need to put forth less effort for communication due to the amount of shared knowledge between them from working on a highly specialized unit of software, This, in turn, enables shared problem solving and improved maintainability of the module (Grant, 1996a; Page-Jones, 1998).

To estimate the level of dependencies across and within modules, we leveraged the leading eigen-vector method to partition the dependency graph of the source code and obtain an estimate of its modularity (Newman, 2006a). The modularity value would be high when cross-module dependencies are low (i.e., loosely coupled) and within-modules dependencies are high, suggesting that the modules are properly partitioned with very little interdependence of tasks across functional units (see Study 1). Therefore, the modularity measure can be used as a proxy that is inversely related to task interdependence.

Procedure

First we extracted the dependency graph for a snapshot of the source code at the beginning of the analysis period. We then used the leading eigen-vector method (Newman, 2006a) to examine how well the graph could be partitioned into sub-graphs. Next, we extracted the modularity measure for the resulting partition from the leading eigen-vector method. We then mean centered the variable since we would be testing an interaction term (Aiken and West, 1991). Finally, we multiplied the value by negative 1 since the modularity estimate was inversely related to task interdependence.

3.4 Controls

3.4.1 Age of Project

The age of the FLOSS project might have an effect on the number of committers since older projects have more established roles and procedures for coordination than younger projects, which might impact performance. In addition, age can serve as a proxy for the stage in which the FLOSS project is, which could affect the level of activity in the project and, thereby, its performance (Stewart et al., 2006). Age was measured in terms of number of months from the time the first committal was made to the source code up to the first day of the analysis period.

3.4.2 Programming Language

Different programming languages follow different philosophies. Some languages, like Python, focus on prototyping and are thus easier to develop for and collaborate on (Stewart et al., 2006; Midha, 2008). Such languages are designed with productivity in mind, which is why we expect language to have an impact on a community's overall development performance.

3.4.3 Project Popularity

A FLOSS project's popularity might also play a role in performance because developers may find popular projects to be more attractive to join due to the social benefits associated with participation and exposure (Lakhani and Wolf, 2007; Lerner and Tirole, 2002). In addition, popular projects tend to have a greater number of software users, which

means the pool of potential contributors to the project is larger (see Study 1). Thus, having more contributors will have an impact on overall project performance.

To measure popularity, we obtained the number of users that reported using the software on ohloh.net. Given that ohloh.net is a social website for FLOSS developers, the number of users that report using a software product serves as a good proxy for popularity amongst developers who are likely to contribute.

3.4.4 Size of Project

Projects with a larger code base are likely to be more complex and have a greater need for change (see Study 1). Therefore, we expect projects to differ in the amount of development activity they require, which might make smaller projects inherently less active. Furthermore, the rate of change in the source code, or the development inertia, has been found to be closely related with the size of the code base in Source Lines of Code (SLOC) (Booch, 2008). Therefore, it was important to control for the size of the code base, estimated as SLOC, for the beginning of the analysis period, which allowed us to compare differently size projects.

3.4.5 Number of Committers

While we are interested in the ratio of committers to external contributors in our study, the total number of committers still has implications for performance. Larger projects can perform a greater number of commits simply by virtue of having a larger group of committers. As a result, we controlled for the number of committers, as we are interested in the effects that are above and beyond size-related variables.

3.5 Analysis and Results

Given the longitudinal nature of our study, we used a mixed-model analysis to fit our statistical models (Cohen et al., 2003). Specifically, we used the lme4 library (Bates and Maechler, 2009) for the GNU R (R Development Core Team, 2009) to perform a Gaussian-based mixed model given the normally distributed dependent variable. Before performing that step, however, we screened our data and made sure that we had no missing data or variables with distributions that violated the assumptions of mixed-model analysis. The correlation matrix (Tab. 3.5) suggests that our variables have discriminant validity given that the correlations between them are low.

Furthermore, we used the influence.ME package (Nieuwenhuis et al., 2009) to assess the influence of our observations on the results of the mixed-model analysis using Cook's d (Cohen et al., 2003). With Cook's d values less than .3, we concluded that no single observation had any excessive influence over the results.

Table 3.6 summarizes the results from our mixed-model analysis (Cohen et al., 2003). The mixed models were fitted such that the observations were nested within projects. Relative to the null model, the controls-only model was able to explain 17.7% of the variability. The main-effects model was a significant improvement over the control with an R^2 of 23.84%. Since we are fitting a Gaussian-mixed model for an unbalanced design, the recommended method to estimate p-values for inference tests is to use a Markov Chain Monte Carlo simulation and estimate the 95% Highest Probability Density Interval (HPD) (Bates and Maechler, 2009; Chen et al., 2000).

Table 3.5: Variable correlations and descriptive statistics

	PERF	CENT	TROUT	CUNC	TINT	AGE	PER	isGPL	isC	isCpp	isPy	POP	SLOC	COM
PERF	1.00													
CENT	0.22	1.00												
TROUT	0.24	0.03	1.00											
CUNC	0.16	0.33	0.10	1.00										
TINT	-0.27	-0.03	0.21	-0.01	1.00									
AGE	0.06	0.11	0.00	0.02	-0.15	1.00								
PER	-0.06	0.03	-0.02	-0.15	-0.01	0.04	1.00							
isGPL	-0.01	-0.01	0.01	0.02	0.00	0.10	0.05	1.00						
isC	0.21	0.12	0.10	0.02	-0.11	0.34	0.00	0.10	1.00					
isCpp	0.20	-0.06	-0.24	-0.06	-0.29	-0.09	-0.01	0.11	0.03	1.00				
isPy	-0.13	-0.05	0.10	0.01	0.20	-0.29	0.00	-0.15	-0.63	-0.32	1.00			
POP	-0.20	-0.13	-0.01	0.00	0.09	-0.39	-0.03	-0.11	-0.18	0.08	0.18	1.00		
SLOC	0.44	0.14	-0.28	-0.02	-0.44	0.32	0.06	-0.04	0.35	0.29	-0.36	-0.26	1.00	
COM	0.66	-0.08	0.14	0.15	-0.22	0.13	-0.01	0.04	0.19	0.17	-0.07	-0.24	0.39	1.00
min	0.00	0.00	-4.64	0.00	-0.42	-7.61	1.00	0.00	0.00	0.00	0.00	-0.21	-6.01	-1.77
mean	4.68	0.48	0.00	0.50	0.00	0.00	4.54	0.42	0.82	0.43	0.14	0.00	0.00	0.00
median	4.84	0.00	0.01	0.00	0.03	0.27	4.00	0.00	1.00	0.00	0.00	0.00	0.01	0.02
max	8.33	1.00	5.03	1.00	0.65	13.16	10.00	1.00	1.00	1.00	1.00	0.15	3.73	3.05
std	1.64	0.00	0.86	0.00	0.29	2.65	2.97	0.00	0.00	0.00	0.00	0.11	1.41	1.15

The null hypotheses that the regression coefficient is different from zero is rejected with an *alpha* < .05 when zero is not within the range of the HPD interval. Based on this inference method, we notice that TROUT has a significant relationship with PERF with a coefficient of 0.2873 in the main-effect model. This result lends support to our hypothesized positive relationship between task routineness and performance in H1. Similarly, the results suggest that TINT ha a significant relationship with PERF with a coefficient of -0.3801. This result lends support to the hypothesized negative relationship between task interdependence and performance in H5. We could not find support for H3 given that the CUNC has a non-significant coefficient.

Table 3.6: Model fitting

Model	Null	Control			Main Effects			Interaction Effects		
		Estimate	HPD Lower	HPD Upper	Estimate	HPD Lower	HPD Upper	Estimate	HPD Lower	HPD Upper
*TROUT*CENT*	—							0.079*	0.007	0.176
*CUNC*CENT*	—							-0.153*	-0.372	-0.03
*TINT*CENT*	—							0.41.	-0.035	0.738
CENT	—				0.601***	0.572	0.772	0.701***	0.649	0.929
TROUT	—				0.287***	0.275	0.378	0.251***	0.224	0.348
CUNC	—				0.099	-0.03	0.151	0.162*	0.023	0.264
TINT	—				-0.37***	-0.813	-0.161	-0.56***	-1.036	-0.294
AGE	—	-0.062***	-0.109	-0.033	-0.071***	-0.113	-0.047	-0.069***	-0.112	-0.045
PER	—	-0.031***	-0.052	-0.019	-0.03***	-0.052	-0.021	-0.031***	-0.052	-0.022
isGPL	—	-0.153	-0.286	0.073	-0.124	-0.233	0.074	-0.125	-0.239	0.073
isC	—	0.228	-0.042	0.544	0.011	-0.285	0.243	0.023	-0.263	0.264
isCpp	—	0.132	-0.082	0.317	0.257*	0.055	0.412	0.252*	0.045	0.403
isPy	—	-0.222	-0.417	0.258	-0.279	-0.475	0.129	-0.271	-0.462	0.135
POP	—	-0.431	-1.666	0.411	-0.353	-1.346	0.472	-0.277	-1.229	0.591
SLOC	—	0.002***	0.067	0.202	0.056***	0.111	0.24	0.06***	0.113	0.24
COM	—	1.226***	0.979	1.115	1.089***	0.878	1.006	1.085***	0.878	1.004
Int	4.652***	4.6779***	4.603	4.77	4.344***	4.25	4.43	4.314***	4.202	4.391
AIC	5538		4623			4308			4311	
LogLik	-2766		-2300			-2138			-2136	
Deviance	5532		4568			4227			4214	
R^2	—		%17.2			%23.01			%23.08	
Chi² Diff	—		983.745			341.096			13.14	
P	—		<.0001***			<.0001***			0.00434***	

Significance codes: '***' <0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Since we used single-indicator variables as proxies for our theoretical constructs, we followed the recommendation of Venkatraman (1989) to test for the hypothesized fit in H2, H4 and H5 as moderation. We also followed the recommendations of Aiken and West (1991) and Cohen et al. (2003) and mean centered all of the continuous variables such that the intercept and lower-order effect terms could be interpreted.

We included the interaction terms in the interaction model between our main effect variables and CENT, which is the moderator that represents the centralization of a FLOSS community's committal structure. Based on the Chi^2 difference test, the interaction model explains a significant amount of variability more than the main effect model. With an R^2 of 20.08% the change in the effect size due to the addition of the interaction terms is quite small. This however is typical of moderation effects in psychological and management studies in which the median for the effect size for studies conducted over the past 30 years was found to be around 0.002 (Aguinis et al., 2005). McClelland and Judd (1993) attribute this to the small amount of residual variance, after accounting for the main effects, that is used to detect moderation effects.

The positive and significant coefficient of the TROUT*CENT coefficient (0.079, p-value < 0.05) lends support to H2, which suggests communities with centralized committal structures are a better fit for routine tasks. To illustrate this, we plotted the simple slopes for the interaction term in Fig. 3.4, which shows how performance increases at a higher rate for centralized committal structures than for decentralized committal structures as the development task becomes more routine (Aiken and West,1991; Cohen et al.,2003). The simple slope for decentralized FLOSS communities is represented by

the main effect coefficient associated with TROUT in the interaction model (0.251, p-value < 0.0001). The significance test for this main effect coefficient is also used to test whether the TROUT simple slope for the decentralized reference group is significantly different from zero (Aiken and West, 1991; Cohen et al., 2003).

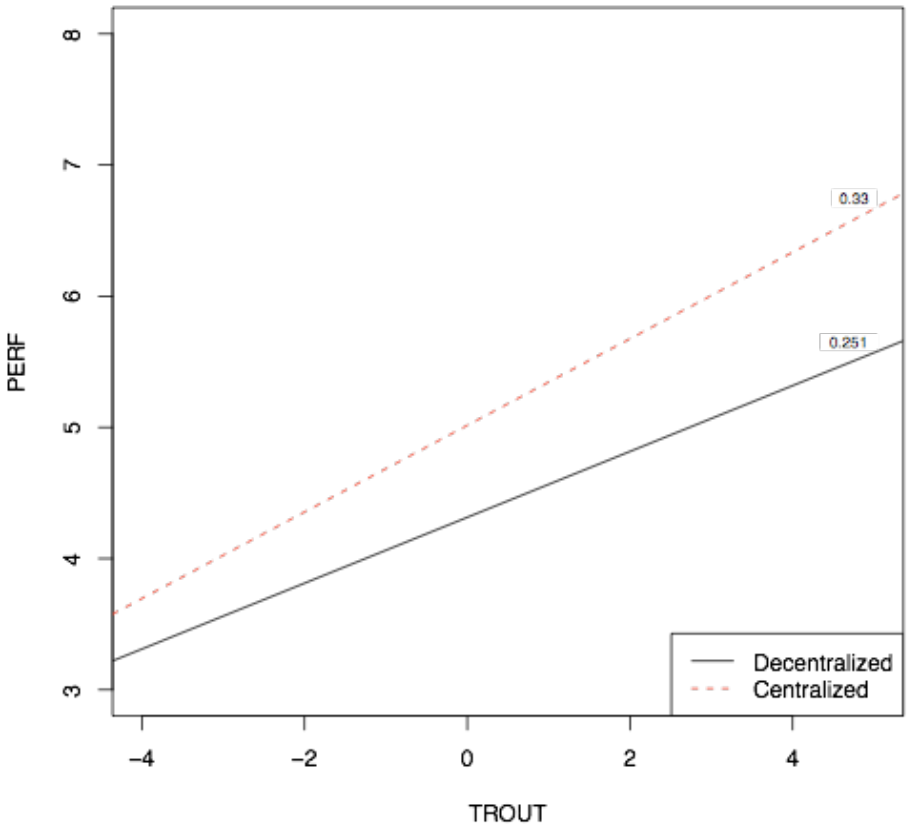

Figure 3.4: Simple slopes for effect of TROUT over different levels of CENT

To obtain the TROUT simple slope for the centralized FLOSS communities, we simply added the TROUT*CENT coefficient to the TROUT coefficient (0.33, p-value = < 0.0001). The significance test for this simple slope was obtained by reverse coding the

CENT variable such that the centralized FLOSS projects are the reference group and then refitting the interaction model (Aiken and West, 1991; Cohen et al., 2003).

As can be seen in Fig 3.4, the higher increase in performance is signified by the steeper and positive slope of the centralized committal structure line. Since TROUT is a continuous and mean-centered variable, the significance of the interaction term TROUT*CENT (p-value < 0.05) was used as a test to confirm that the difference in the simple slopes between the two levels of CENT is indeed significant (Aiken and West, 1991).

The negative and significant slope of the CUNC*CENT coefficient (-0.147, p-value < 0.05) lends support to H4, which suggests that a decentralized committal structure is a better fit for dealing with higher contributor uncertainty (i.e., the increase in numbers of new contributors). This result also confirms that the simple slopes for centralized and decentralized FLOSS communities are significantly different (Aiken and West, 1991).

To illustrate this difference, we plotted the simple slopes for the interaction term as shown in Fig. 3.5. The graph shows that the effect of uncertainty on performance has a positive and steeper slope for decentralized communities than for centralized communities. The CUNC simple slope for centralized FLOSS communities is non-significant (0.009, p-value > 0.1)[4], suggesting that contributor uncertainty has no impact on performance for centralized FLOSS communities. On the other hand, the CUNC simple slope for

[4] The significance test was obtained after reverse coding CENT and refitting the interaction model and using the significance test for the CUNC coefficient (Aiken and West, 1991).

decentralized communities was positive and significant (0.162, p-value < .05), which is contrary to our expectations.

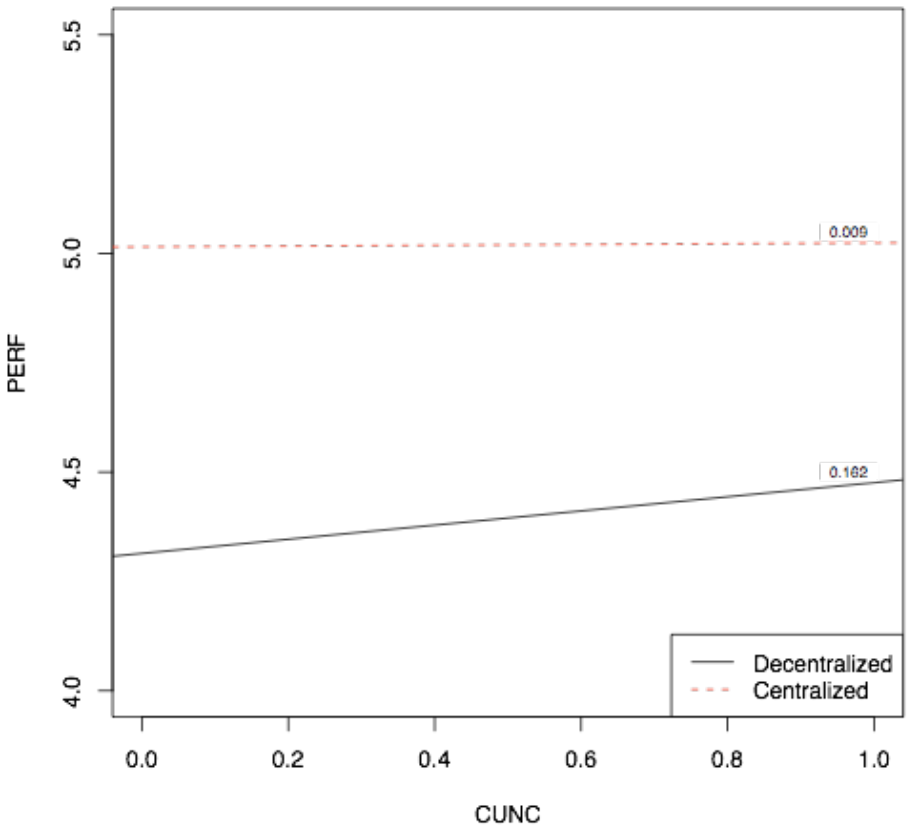

Figure 3.5: Simple slopes for effect of CUNC over different levels of CENT

Finally, we found partial support for H6, which suggests that centralized committal structures are a better fit when there is a high degree of task interdependence, as illustrated by the coefficient of the TINT*CENT term (0.406, p-value < 0.1). The results suggest that FLOSS communities with centralized committal structures can perform better

than communities with decentralized committal structures as task interdependence is increased.

To illustrate this, we plotted the simple slopes for the interaction term shown in Fig. 3.6. The steeper slope for the relationship between TINT and PERF for FLOSS communities with decentralized committal structures suggests that centralized committal structures are a better fit for increasing task interdependence, as performance drops at a much slower rate than in decentralized committal structure (-0.5603, p-value = < 0.0001)[5]. For the simple slope for centralized committal structures, however, there was only partial support that it was significantly different from zero (-0.151, p-value < 0.1), suggesting that interdependence might have a less detrimental effect on performance for centralized FLOSS communities.

Furthermore, when looking at the increasing performance from the right side of the graph to the left, the same graph suggests that decentralized committal structures can observe higher performance gains as task interdependence is reduced by means of improving the software design to be more modular. However, the difference in between these simple slopes only finds partial support with the partially significant TINT*CENT coefficient (-0.41, p-value < 0.1) (Aiken and West, 1991).

[5] The significance test was obtained after reverse coding CENT and refitting the interaction model and using the significance test for the TINT coefficient (Aiken and West, 1991).

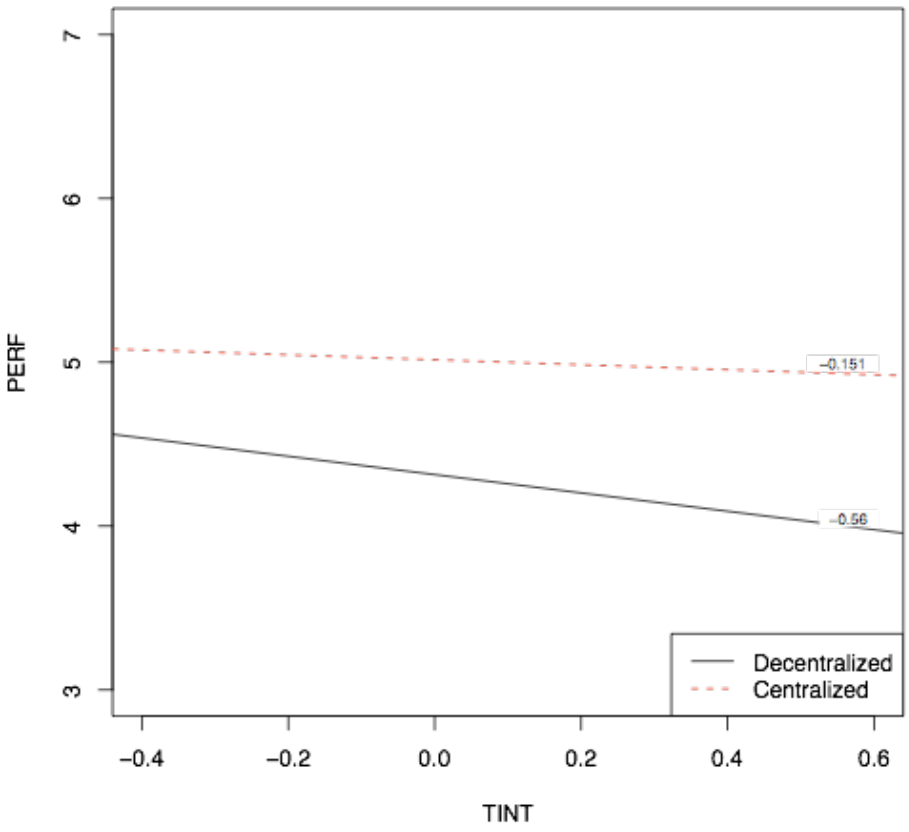

Figure 3.6: Simple slopes for effect of TINT over different levels of CENT

4 Discussion

The main premise of our theory is that development activities within a FLOSS community are information-processing tasks (Galbraith, 1973). Under the assumption of bounded rationality, we forwarded the argument that the committal structure will be the main bottleneck of performance as the information-processing requirements of a FLOSS community increased. The results of our statistical analysis suggest that there is good overall support for the theory we have forwarded about the performance of FLOSS com-

munities as summarized in Tab. 3.7. We also summarize the main contributions of our work in Tab. 3.8

Table 3.7: Summary of empirical findings from the higher-order model

Hypothesis	Coeffecient	Support
H1: Task routineness is positively related to a FLOSS community's performance.	*TROUT*: 0.287 p-value < 0.001	Supported.
H2: As tasks become more routine, FLOSS communities with centralized committal structures will have higher performance gains than communities with decentralized committal structures.	*TROUT*CENT*: 0.079 p-value < 0.001	Supported.
H3: Contributor uncertainty will be positively related to a FLOSS community's performance.	*CUNC*: 0.099 p-value > 0.1	Not supported.
H4: As contributor uncertainty increases, FLOSS communities with centralized committal structures will have lower performance gains than communities with decentralized committal structures.	*CUNC*CENT*: -0.147 p-value < 0.05	Supported.
H5: Task interdependency will be negatively related to a FLOSS community's performance.	*TINT*: -0.37 p-value < 0.001	Supported.
H6: As task interdependency is increased, FLOSS communities with centralized committal structures will have higher performance gains than communities with decentralized committal structures.	*TINT*CENT*: 0.41 p-value < 0.1	Partially supported.

Table 3.8: Summary of this work's main contributions

Finding	Impact
FLOSS communities performing simple tasks perform better.	• Importance of good source code design to simplify development. • Contributors should work closer to the community and make small incremental changes rather than work in isolation and accumulate their patches into a single patch that is hard to incorporate.
Centralized committal structures are a better fit for routine tasks.	• The committal structure should match the needs of the community. • There is no single superior structure.
Decentralized committal structures are a better fit under high contributor uncertainty.	• Decentralized committal structures are necessary if community involvement is valued. • Brooks' law is not obsolete; the committal structure has to be overwhelmed for it to become obvious.
Task interdependence increases information-processing requirements for a FLOSS community and reduces performance.	• Importance of modularizing the source code and its effect on the performance of a FLOSS community. • Further validation of the Newman (2006a) modularity measure.
Decentralized committal structures are a better fit under conditions of low task interdependence.	• Decentralized committal structures are enabled by proper code design. • Centralized committal structures might be the only way to continue to maintain and develop tightly coupled code bases. • Brooks' and Raymond's views are complementary. Raymond explains how FLOSS development is conducted under conditions of fit, while Brooks' views become apparent under condition with lack of fit.

In H1, we hypothesized that FLOSS communities will perform better when their development tasks are more routine. We attributed the improved performance to the reduced information-processing requirement of routine tasks, which are less likely to overwhelm the committal structure. The results from our statistical analysis lend support to this hypothesis.

These findings suggest that FLOSS communities could improve their performance if development tasks are simplified. Specifically, communities should invest their time in properly designing and organizing the source code such that similar functionality is encapsulated in specific modules (Page-Jones, 1998). The findings also signify the importance of working within the community for contributors. By being close to the community and contributing small and incremental changes, contributors are likely to find the committal structure more responsive. Working in isolation and contributing a large patch to the community will likely require a significant time investment from committers and will most likely result in the patch not being accepted.

In addition, we hypothesized that centralized committal structures are a better fit for routine development tasks because the coordination requirements between committers are generally lower for centralized committal structures than for decentralized structures due to the smaller number of communication channels that each committer is required to maintain with other committers (Brooks, 1975). This hypothesis also found support in our statistical analysis.

What these results suggest is that even though decentralized committal structures might have a higher capacity to process information, they are not the best fit for every

situation. Communities that maintain projects that require routine development activities, either from the simplicity of the problem or the maturity of project with plateaued development activity, could be developed and maintained more efficiently with a centralized committal structure. On the other hand, more complex projects could benefit greatly from the distributing the development process. However, there is the limitation that the design of the code base must be improved to enable such a distributed development mode, as we shall explain in the discussion about the results for H5 and H6.

We hypothesize in H3 that increased uncertainty from having new contributors participate might have a detrimental effect on performance, as new contributors, and even their patches, will demand extra attention from committers. The results did not lend support to this hypothesis. It is possible that the dichotomization of the variable was the cause of our failure to detect such an effect, which we acknowledge to be an unavoidable limitation in our methods (MacCallum et al., 2002). It could also mean that the centralized committal structure is overwhelmed and that the committers are doing a good job of organizing the committal activities so as to continue to maintain progress. To confirm that this was indeed the case, we would need to examine the number of ignored patches by the community. While there is no denying that such data could enrich our methods, we found it very difficult to collect, which seems to be a limitation that could be addressed in future work.

We were surprised, however, from the significant and negative interaction term that suggests, as we predicted in H4, that FLOSS communities with decentralized committal structures tend to perform better under higher uncertainty. The simple slopes suggest that

uncertainty had no impact on performance for centralized committal structures. However, higher uncertainty translated into higher performance under decentralized committal structures.

The higher performance of decentralized committal structures under high uncertainty could be attributed to decentralized committal structures being able to deal with the external uncertainty introduced by new contributors more effectively. As we explained in Sec. 3.2.2, committal activities are performed in parallel, resulting in a continuance of committal activity when new contributors require the attention of one of the committers. This may even lead to more of the new contributors turning into regular contributors and adding to the increased performance of the community.

What is interesting about this finding is that it shows the direct tension between Brooks' and Raymond's views. The lack of performance improvement for centralized structures shows, as Brooks' had anticipated, that there are no performance gains. To take advantage of new contributors' effort, the community should be able to decentralize the committal structure to be able to handle the extra uncertainty introduced by new contributors.

In light of these results, we conclude that it is a bit premature to consider Brooks' law obsolete. Rather, it seems that Brooks' law starts to take effect in FLOSS communities when the committal structure hits the limits of its information-processing capacity. This is even further highlighted by the support for H4 in which we specifically hypothesized that decentralized committal structures are a better fit for dealing with contributor uncertainty. We believe this to be true mainly due to their greater capacity for information processing

from their distribution of tasks and requisite development knowledge among a larger group of committers, which reduces the workload on any single individual.

In H5, we hypothesized that communities with fewer interdependencies in their development task will generally perform better. We attribute this improved performance to the reduction in such communities' need for coordination, which frees up more of the committal structure's information-processing capacity for performing more tasks. The need from coordination stems mainly from dependencies between the source files on which different developers work, which creates the need for coordination Crowston (1997). By improving the design of the source code to exhibit higher modularity (i.e., low coupling between modules and high cohesion within a module), FLOSS communities can improve their performance. The results from our analysis generally support this hypothesis. The importance of this finding also stems from the fact that it offers further validation to the modularity measure (Newman, 2006a).

Finally, we hypothesize in H6 that centralized committal structures are a better fit for highly interdependent tasks because highly interdependent tasks will require greater coordination efforts. In communities with decentralized committal structures, committers will have to maintain communication channels with a larger group of committers to maintain the functional integrity of the developed software (Brooks, 1975; Crowston, 1997). With higher task interdependence, the amount of information that needs to be exchanged in each channel will increase, making it impossible for a committer to maintain the same amount of channels. Therefore, centralized structures have the advantage of having a smaller group of committers maintaining a smaller number of communication

channels, which, in turn, take up much less of the committal structure's information-processing capacity. This hypothesis found support with the higher decrease in performance for decentralized committal structures as compared to centralized committal structures.

The flip-side of the previous argument is that the improvement gains to software design in decentralized committal structures are greater than for centralized committal structures. We would even go as far as saying that large FLOSS communities with large committal structures require a highly modular design to continue to function. This also goes back to the tension between Brooks' and Raymond's view. While Raymond supported the distributed development model of the FLOSS community and touted its benefits, the caveat to this idea is that the software should be designed to enable such a process. Adding developers or improving performance always comes at an information-processing cost, and FLOSS communities will be able to improve on both only up to what the information-processing capacity of the committal structure would allow them to. After that point, the community must make a trade-off to either reduce their information-processing needs or increase their information-processing capacity.

5 Limitations and Future Research

Although our work provides insight into the development structure of FLOSS communities, care should be taken so as not to confuse the development structure with the communication structure (Crowston and Howison, 2005). There is evidence that a FLOSS community's communication structure will not match its development structure (Mockus

et al., 2002). As such, it will be rather interesting to know how the two relate. Our work raises the following questions that future studies could help address: could an increase in communication activity signal that there is a problem in coordination that could be remedied by increased communication; and will this affect the sustainability of the community since it would result in an increased cost for participation and an information overload on the members (Jones et al., 2004; Kuk, 2006)?

The way we conceptualized the development structure to exclude the communication structure might seem to be a limitation; however, it was a necessary simplification that lead us one step closer to understanding organizational structures in FLOSS communities. We explained that we took this approach because we saw the main function of a FLOSS community as being the development of software and that all contributions must be processed by committers; therefore, they are indeed a bottleneck in the development structure.

Prior work that examined development organization in FLOSS communities took a small-sample approach (Mockus et al., 2002; Shah, 2006; O'Mahony and Ferraro, 2007) or mainly examined the communication structure (Crowston and Howison, 2005). This work came as a response to the call by Koch (2004) for more studies that attempt to generalize findings across a diverse set of FLOSS projects. Future studies could build on our work and take a more granular approach to classifying FLOSS community structures and understanding their effects on participation and productivity in addition to their relation to the communication structure.

It is also important to highlight the limitations of our methods in the hope that future studies could improve upon them. The first of these limitations relates to the quarterly

window used to aggregate our data. While we have justified our selection based on prior work that used anywhere from a one-month to a one-year window, we feel that our selection was somewhat arbitrary. We hope to vary this window of analysis with our future work and compare the results. The limited computational resources at this time prevented us from feasibly performing this task.

Furthermore, due to the unusual distribution of some of our variables, we were left with no choice but to dichotomize the variables. Such an approach might yield spurious relationships and result in lowering the power of the statistical analysis (MacCallum et al., 2002). However, we ruled out the chance that our results are spurious by conducting the analysis using alternative methods for factoring the variables and found the results to hold (see Appendix A). We chose to use the results of the dichotomized variables because they are easier to present and interpret. The fact that we found significant results despite the lower statistical power of our chosen method of factorization suggests that the actual effects that we are trying to detect might have a larger effect size in reality. While we were limited in terms of the variables we could extract from the available data archive, we hope to make improvements in future work by augmenting our data with surveys and alternative data sources.

Finally, while we may have espoused the idea that there is no single superior organizational form, we only examined a subset of the potential tradeoffs between the different committal structures. There are still some broader implications for the different committal structures that future studies could expand on. For example, what are the actual organizational costs for a community to change the committal structure and how might that

change impact contributors and performance over the transition period. Furthermore, given that such initiatives could be conceptualized as a form of business-process reengineering, the risk that such initiatives could fail does exist. Therefore, it will be important to understand what factors might contribute to the success or failure of such initiatives.

6 Conclusion

The goal we set out to achieve with this study is to determine if organizational theories of fit apply to FLOSS communities, knowing that FLOSS communities have emergent structures by nature (O'Mahony and Ferraro, 2007). We leveraged OIPT (Galbraith, 1973) to argue that FLOSS communities that perform highly in terms of source code output will organize to achieve the best fit between their information-processing capabilities and needs. The empirical results lend support to our theory that high-performing FLOSS communities match their development structure, as the embodiment of their information-processing capabilities, with their software development tasks, as the main source of their information–processing needs.

This work contributes to both theory and practice. From a theoretical standpoint, we have shown that FLOSS communities are not so different from conventional organizational units from an information-processing perspective. We have also given an explanation as to how the emergent development structure in FLOSS communities could form in response to the community's information-processing needs. In addition, we have given an account of the different development-related organizational structures and have found

that there is no single superior structure; rather, what is important in the development process is finding the fit between the community's development needs and its structure.

Our theoretical deductions and empirical results support the idea that FLOSS communities performing development tasks that are generally routine, highly interdependent, and generate little contributor uncertainty will perform better under a centralized committal structure. On the other hand, decentralized committal structures thrive under the conditions of task non-routineness, low task interdependence, and high contributor uncertainty.

More importantly, these findings highlight the tension between the views forwarded by Brooks (1975) about software development and Raymond (2001) about FLOSS development. While both of these views have been seen as conflicting, we show with this work that they are actually complementary. We attribute the conflict to the lack of clarity in the assumptions of both views and find that Raymond's view generally holds true of FLOSS development until the committal structure is overwhelmed. In such cases, we begin to see signs that Brooks' views hold true even in FLOSS development.

The results of our study are equally useful to FLOSS community organizers and organizations that seek to increase the development output from FLOSS communities that are under their management. We have highlighted the importance of good code design in managing dependencies between developers to allow for a more distributed development process.

Our work also makes clear that promoting committers is not always a favorable option to improve the output of the community or to help developers when they are over-

whelmed, especially when the source code is highly interdependent. The best course of action in such situations involves a reorganization of the source code. Adding more committers might simply heighten the cost of coordination between developers, thereby reducing the overall performance of the community. This shows how FLOSS communities are no different than any other software development team and that Brooks' law (Brooks, 1975), as one of the most important classical theories on software project management, still holds true in the FLOSS context.

Appendices

Appendix A: Robustness of Median Split Results

To ensure that our results are not caused by the median split (MacCallum et al., 2002), we performed a tertile split on both CUNC and CENT and refit our interaction model. The results are summarized in Tab. 9 and 10, which show that our results hold, suggesting that the median split did not have an impact on our results. With all continuous variables mean centered, we could interpret the interaction effects and lower order main effects (Aiken and West, 1991).

If you recall from Tab. 7, we found support for H1, H2, H4, and H5. In Tab. 9, we can see that TROUT has a positive and significant coefficient, thus supporting H1. TINT also has a negative and significant coefficient, providing support for H5. In Tab. 9, we have the results from the interaction model. The $\text{TROUT*CENT}_{\text{hi}}$ coefficient is positive and significant suggesting that the TROUT coefficient for the high centralization group is higher and significantly different than the coefficient for the reference low centralization group, thus providing support for H2. The CUNC_{hi}*CENT coefficients are also significant or partially significant and negative, suggesting that groups with high centralization have a significantly lower coefficient than the reference group with low centralization, providing support for H4.

Table 3.9: Main-effect model with tertile split of CENT and CUNC

Term	Coeffecient	P-Value	Supports
TROUT	0.287	0.0001***	H1
CUNC_med	0.13	0.296	
$CUNC_{hi}$	0.103	0.575	
TINT	-0.43	0.0001***	H5
$CENT_{med}$	0.336	0.0001***	
$CENT_{hi}$	0.857	0.0001***	

Table 3.10: Interaction model with tertile split of CENT and CUNC

Term	Coeffecient	P-Value	Supports
TROUT*$CENT_{med}$	-0.018	0.315	
TROUT*$CENT_{hi}$	0.11	0.039*	H2
$CUNC_{med}$*$CENT_{med}$	-0.129	0.315	
$CUNC_{med}$*$CENT_{hi}$	0.164	0.513	
$CUNC_{hi}$*$CENT_{med}$	-0.336	0.002**	H4
$CUNC_{hi}$*$CENT_{hi}$	-0.106	0.096.	H4
TINT*$CENT_{med}$	0.347	0.301	
TINT*$CENT_{hi}$	0.241	0.844	
TROUT	0.264	0.0001***	
CUNC_med	0.097	0.75	
$CUNC_{hi}$	0.259	0.006**	
TINT	-0.642	0.001***	
$CENT_{med}$	0.499	0.0001***	
$CENT_{hi}$	0.796	0.0001***	

References

Aguinis, H., Beaty, J. C., Boik, R. J., & Pierce, C. A. (2005). Effect Size and Power in Assessing Moderating Effects of Categorical Variables Using Multiple Regression: A 30-Year Review. *Journal of Applied Psychology*, 90(1), 94-107.

Ahuja, M. K. and Carley, K. M. (1999). Network structure in virtual organizations. *Organization Science*, 10(6):741-757.

Aiken, L. S. and West, S. G. (1991). *Multiple regression: Testing and interpreting interactions*. Sage Publications, Inc.

AlMarzouq, M., Zheng, L., Rong, G., and Grover, V. (2005). Open source: Concepts, benefits, and challenges. *Communications of AIS*, 2005(16):756-784.

Baldwin, C. Y. and Clark, K. B. (2000). *Design rules, Vol. 1: The power of modularity*. The MIT Press.

Baldwin, C. Y. and Clark, K. B. (2006). The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116-1127.

Bates, D. and Maechler, M. (2009). *lme4: Linear mixed-effects models using S4 classes*. R package version 0.999375-31.

Blau, P. M. (1968). The Hierarchy of Authority in Organizations. *The American Journal of Sociology*, 73(4), 453-467.

Bonaccorsi, A. and Rossi, C. (2003). Why open source software can succeed. *Research Policy*, 32(7):1243-1258.

Booch, G. (2008). Measuring architectural complexity. *IEEE Softw.*, 25(4):14-15.

Brooks, F. (1975). The mythical man-month. In *Proceedings of the International Conference on Reliable Software*, volume 10. ACM Press.

Capiluppi, A. and Adams, P. J. (2009). Reassessing Brooks' law for the free software community. In Boldyreff, C., Crowston, K., Lundell, B., and Wasserman, A. I., editors, *OSS*, volume 299 of *IFIP*, pages 274-283. Springer.

Chen, M.-H., Shao, Q.-M., and Ibrahim, J. G. (2000). *Monte Carlo methods in bayesian computation (Springer series in statistics)*. Springer.

Choudhury, V. and Sampler, J. L. (1997). Information specificity and environmental scanning: An economic perspective. *MIS Quarterly*, 21(1):25-53.

Cohen, J., Cohen, P., West, S., and Aiken, L. (2003). *Applied Multiple Regression/ Correlation Analysis for the Behavioral Sciences*. Lawrence Erlbaum, third edition.

Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science*, 8(2):157-175.

Crowston, K., Annabi, H., and Howison, J. (2003). Defining open source software project success. In *Proceedings of the 24th International Conference on Information Systems (ICIS 2003*, pages 327-340.

Crowston, K. and Howison, J. (2005). The social structure of free and open source software development. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/1207.

Crowston, K., Wei, K., Li, Q., Eseryel, U., and Howison, J. (2005). Coordination of free/ libre open source software development. In *ICIS 2005 Proceedings*.

Daft, R. L. and Lengel, R. H. (1986). Organizational information requirements, media richness and structural design. *Management Science*, 32(5):554-571.

DeLone, W. H. and McLean, E. R. (1992). Information systems success: The quest for the dependent variable. *Information Systems Research*, 3(1):60-95.

Espinosa, J. A., Slaughter, S. A., Kraut, R. E., and Herbsleb, J. D. (2007). Familiarity, complexity, and team performance in geographically distributed software development. *Organization Science*, 18(4):613-630.

Fichman, R. G. and Kemerer, C. F. (1997). The assimilation of software process innovations: An organizational learning perspective. *Management Science*, 43(10):1345-1363.

Fitzgerald, B. (2006). The transformation of open source software. *MIS Quarterly*, 30(3):587-598.

Freeman, L. (1979). Centrality in social networks conceptual clarification. *Social Networks*, 1(3), 239, 215.

Galbraith, J. R. (1973). *Designing complex organizations*. Addison-Wesley series on organization development. Addison Wesley.

Gemino, A., Reich, B. H., and Sauer R, C. (2007). A temporal model of information technology project performance. *Journal of Management Information Systems*, 24(3):9-44.

Goldratt, E. M. and Cox, J. (1994). *The goal*. North River Press, second edition.

Grant, R. M. (1996a). Prospering in dynamically-competitive environments: Organizational capability as knowledge integration. *Organization Science*, 7(4):375-387.

Grant, R. M. (1996b). Toward a knowledge-based theory of the firm. *Strategic Management Journal*, 17:109-122.

Grewal, R., Lilien, G. L., and Mallapragada, G. (2006). Location, location, location: How network embeddedness affects project success in open source systems. *Management Science*, 52(7):1043-1056.

Hage, J., & Aiken, M. (1967). Relationship of Centralization to Other Structural Properties. *Administrative Science Quarterly*, 12(1), 72-92.

Howison, J. and Crowston, K. (2004). The perils and pitfalls of mining sourceforge. In *Proceedings of the International Workshop on Mining Software Repositories (MSR 2004*, pages 7-11.

Jones, Q., Ravid, G., and Rafaeli, S. (2004). Information overload and the message dynamics of online interaction spaces: A theoretical model and empirical exploration. *Information Systems Research*, 15(2):194-210.

Koch, S. (2004). Profiling an open source project ecology and its programmers. *Electronic Markets*, 14(2):77-88.

Koch, S. and Schneider, G. (2002). Effort, co-operation and co-ordination in an open source software project: Gnome. *Information Systems Journal*, 12(1):27-42.

Kogut, B. and Zander, U. (1992). Knowledge of the firm, combinative capabilities, and the replications of technology. *Organization Science*, 3(3):383-397.

Krishnamurthy, S. (2002). Cave or community? An empirical examination of 100 mature open source projects. http://firstmonday.org/htbin/cgiwrap/bin/ojs/index.php/fm/article/view/960/881.

Kroah-Hartman, G., Corbet, J., and McPherson, A. (2008). Linux Kernel development (April 2008). https://www.linuxfoundation.org/publications/linuxKerneldevelopment.php.

Krogh, G. V. and Hippel, E. V. (2006). The promise of research on open source software. *Management Science*, 52(7):975-983.

Kuk, G. (2006). Strategic interaction and knowledge sharing in the KDE developer mailing list. *Management Science*, 52(7):1031-1042.

Lakhani, K. R. and Wolf, R. G. (2007). *Why hackers do what they do: Understanding motivation and effort in free/open source software projects*, pages 3-22. The MIT Press.

Lawrence, P. R. and Lorsch, J. W. (1967). Differentiation and integration in complex organizations. *Administrative Science Quarterly*, 12(1):1-47.

Lee, G. K. and Cole, R. E. (2003). From a firm-based to a community-based model of knowledge creation: The case of the linux Kernel development. *Organization Science*, 14(6):633-649.

Lerner, J. and Tirole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics*, 50(2):197.

Liu, X. and Iyer, B. (2007). Design architecture, developer networks and performance of open source software projects. In *ICIS 2007 Proceedings*.

MacCallum, R. C., Zhang, S., Preacher, K. J., and Rucker, D. D. (2002). On the practice of dichotomization of quantitative variables. *Psychological Methods*, 7(1):19-40.

MacCormack, A., Rusnak, J., and Baldwin, C. Y. (2006). Exploring the structure of complex software designs: An empirical study of open source and proprietary code. *Management Science*, 52(7):1015-1030.

Malone, T. W. and Crowston, K. (1994). The interdisciplinary study of coordination. *ACM Comput. Surv.*, 26(1):87-119.

March, J. G. and Simon, H. A. (1993). *Organizations*. Blackwell Publishers.

McClelland, G. H., & Judd, C. M. (1993). Statistical difficulties of detecting interactions and moderator effects. *Psychological Bulletin*, 114(2), 376-390.

Midha, V. (2008). Does complexity matter? The impact of change in structural complexity on software maintenance and new developers' contributions in open source software. In *ICIS 2008 Proceedings*.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Softw. Eng. Methodol.*, 11(3):309-346.

Newman, M. E. J. (2006a). Finding community structure in networks using the eigenvectors of matrices. *Physical Review E*, 74:036104.

Newman, M. E. J. (2006b). Modularity and community structure in networks. *PNAS*, 103:8577.

Newman, M. E. J. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical Review E*, 69:026113.

Nidumolu, S. (1995). The effect of coordination and uncertainty on software project performance: Residual performance risk as an intervening variable. *Information Systems Research*, 6(3):191-219.

Nieuwenhuis, R., Pelzer, B., and te Grotenhuis, M. (2009). *influence.ME: Tools for detecting influential data in mixed effects models*. R package version 0.7.

Oh, W. and Jeon, S. (2007). Membership herding and network stability in the open source community: The ising perspective. *Management Science*, 53(7):1086-1101.

O'Mahony, S. and Bechky, B. A. (2008). Boundary organizations: Enabling collaboration among unexpected allies. *Administrative Science Quarterly*, 53(3):422-459.

O'Mahony, S. and Ferraro, F. (2007). The emergence of governance in an open source community. *Academy of Management Journal*, 50(5):1079-1106.

Ouchi, W. G., & Dowling, J. B. (1974). Defining the Span of Control. Administrative Science Quarterly, 19(3), 357-365.

Page-Jones, M. (1998). Cohesion. In The Practical Guide to Structured Systems Design. Retrieved October 6, 2008, from http://www.waysys.com/ws_content_bl_pgssd_ch06.html

Paul, R. (2009a). SourceForge adds support for new version control systems. http://arstechnica.com/open-source/news/2009/03/sourceforge-adds-support-for-new-version-control-systems.ars.

Paul, R. (2009b). SourceForge wants to be collaboration powerhouse, buys ohloh. http://arstechnica.com/open-source/news/2009/05/sourceforge-acquires-foss-code-metric-web-site-ohloh.ars.

Perrow, C. (1967). A framework for the comparative analysis of organizations. *American Sociological Review*, 32(2):194-208.

R Development Core Team (2009). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0.

Raymond, E. (2001). *The cathedral and the bazaar: Musings on Linux and open source by an accidental revolutionary*. O'Reilly, Cambridge, MA, revised edition.

Riehle, D. (2007). The economic motivation of open source software: Stakeholder perspectives. *Computer*, 40(4):25-32.

Sanchez, R. and Mahoney, J. (1996). Modularity, flexibility, and knowledge management in product and organization design. *Strategic Management Journal*, 17:76, 63.

Scacchi, W. (2002). Understanding the requirements for developing open source software systems. *Software, IEE Proceedings*, 149(1):24-39.

Schweik, C. M., English, R. C., Kitsing, M., and Haire, S. (2008). Brooks' versus Linus' law: an empirical test of open source projects. In *dg.o '08: Proceedings of the 2008 international conference on digital government research*, pages 423-424. Digital Government Society of North America.

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

Simon, H. A. (1955). A behavioral model of rational choice. *The Quarterly Journal of Economics*, 69(1):99-118.

Simon, H. (1997). Administrative Behavior, 4th Edition. Free Press.

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.

Stewart, K. J. and Gosain, S. (2006). The impact of ideology on effectiveness in open source software development teams. *MIS Quarterly*, 30(2):291-314.

Tan, Y., Mookerjee, V., and Singh, P. (2007). Social capital, structural holes and team composition: Collaborative networks of the open source software community. *In ICIS 2007 Proceedings*.

Tiwana, A. (2008). Does interfirm modularity complement ignorance? A field study of software outsourcing alliances. *Strategic Management Journal*, 29(11):1252, 1241.

Turner, K. L. and Makhija, M. V. (2006). The role of organizational controls in managing knowledge. *Academy of Management Review*, 31(1):197-217.

Tushman, M. L. (1979). Work characteristics and subunit communication structure: A contingency analysis. *Administrative Science Quarterly*, 24(1):82-98.

Tushman, M. L. and Nadler, D. A. (1978). Information processing as an integrating concept in organizational design. *Academy of Management Review*, 3(3):613-624.

Venkatraman, N. (1989). The concept of fit in strategy research: Toward verbal and statistical correspondence. *Academy of Management Review*, 14(3):423- 444.

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).

Wu, J., Goh, K.-Y., and Tang, Q. (2007). Investigating success of open source software projects: A social network perspective. In *ICIS 2007 Proceedings*.

Zander, U. and Kogut, B. (1995). Knowledge and the speed of the transfer and imitation of organizational capabilities: An empirical test. *Organization Science*, 6(1):76- 92.