Tapping into the source: corporate involvement in open source software

Jan Eilhard

École doctorale n°396 : Economie, organisation, société

Doctorat ParisTech

THÈSE

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Economie et finance »

présentée et soutenue publiquement par

Jan EILHARD

le 14 mai 2010

Tapping Into the Source : Corporate involvement in open source software

Directeur de thèse : **François LEVEQUE**
Co-encadrement de la thèse : **Yann MENIERE**

THÈSE

Jury

M. Marc BOURREAU, Professeur, SES, Paris Telecom - Paristech	Rapporteur
M. Eric BROUSSEAU, Professeur, Economix, Université de Paris X	Examinateur
M. Eric STROBL, Professeur, CECO, Ecole Polytechnique	Examinateur
M. Mikko VALIMAKI, Professeur, Helsinki University of Technology	Rapporteur
M. François LEVEQUE, Professeur, CERNA, Ecole des Mines - Paristech	Directeur de thèse

MINES ParisTech
CERNA
60 boulevard Saint Michel 75272 PARIS cedex 06

L'implication des entreprises aux logiciels libres

Résumé : La participation des entreprises aux logiciels libres touche des domaines différents aux sciences économiques et sciences sociales. Elle est parmi d'autres une expérience naturelle pour la production des biens publics, pour l'innovation collective, pour les technologies disruptives, pour l'externalisation des technologies ou pour les organisation décentralisée. Cette thèse se concentre sur la production des biens publics, l'innovation collective et l'externalisation des technologies. Dans notre analyse, nous utilisons une base de données de 10,000 logiciels libres trouvable sur SourceForge et lions l'information des développeurs aux profiles académiques, salariés et bénévoles.
Mots clés : Logiciel libre, innovation ouverte, économétrie

Tapping Into the Source: Corporate involvement in open source software

Abstract: Corporate participation in open source software touches upon a number of different issues in economics and social sciences. It is, among others, a natural experiment for the private provision of a public good, for gift giving, for collective invention, for disruptive technologies, for technology outsourcing or for decentralized organizations. Little does it surprise then that academia strives to understand corporate involvement in open source software. This thesis focuses on three aspects within this vast field: the private provision of a public good, collective invention as well as outsourcing of technology. For our analysis, we use a dataset of 10,000 open source applications from SourceForge and relate information about developers to three categories as academic, corporate or private ones.
Keywords: Open source software, open innovation, econometrics

Acknowledgements

I am grateful to Professor Lévêque for his support and guidance through this intellectual adventure. I would also like to thank Yann Ménière without whom this work would not have been possible. Working with the two of them has been a great pleasure.

I thank Professor Strobl for his comments on my dissertation and earlier versions of my working papers. I am grateful to Professor Bourreau, Professor Brousseau and Professor Välimäki for their remarks on the thesis.

I would like to thank Professor Glachant for organizing the PhD seminars and providing invaluable feedback on my papers.

I thank all PhD students at CERNA for making work fun, in particular Antoine Dechezlepretre, Timothée Olivier, Henry Delcamp and Benjamin Bureau. I would especially like to thank Stéphanie Fen Chong and Yéhoudit Cohen for sharing the office and their friendship.

I would like to thank Sésaria Ferreira for organizing my PhD defense and for helping me tackle all administrative hurdles.

I thank the staff of the Berkman Center of Internet and Society for their hospitality.

On a personal note, I thank my parents and my family for everything. Even if it sounds trite, it's true that I owe everything to them.

Thank you.

La nécessité est, d'ailleurs, de tous les maîtres,

celui qu'on écoute le plus et qui enseigne le mieux.

— Jules Verne (L'île mystérieuse)

Contents

List of Figures

List of Tables

Preface

- Chapter 1 : "Corporate Open Source: A Literature Review" is an extract of the report "Open Source Incorporated" (January 2009) available on SSRN: http://ssrn.com/abstract=1360604
- Early drafts of Chapter 2 : "A Look Inside the Forge: Developer productivity and spillovers in Sourceforge projects" were presented at
  - the "Economics and Econometrics of Innovation" seminar in Paris in December 2008,
  - the MERIT-UNU seminar in Maastricht 2009,
  - the International Industrial Organization Conference in Boston, 2009,
  - the FLOSSworkshop 2009 in Padova, 2009.
- Drafts of Chapter 3 : "Loose Contracts, Tight Control: Corporate contributions in SourceForge projects" were presented at
  - the International Industrial Organization Conference in Washington, 2008,
  - the Berkman Center in Cambridge, MA, 2008

Introduction

Open source software was born out of necessity. As the anecdote goes, Richard Stallman conceived the idea for free/open source software because he did not want to walk to a shared printer for nothing. When he wanted to add a small computer program to the printer that would notify him when a print was finished, he was barred access to the software of the printer. Frustated by the state of commercial, proprietary software, he set out to do things differently. Richard Stallman drafted the first version of the General Purpose License, the first open source license, in 1989. Its principles were simple. Anyone can modify and redistribute software licensed under the GPL as long as they credit the original developer and allow others to do the same. This premise turns software into a public good, non-excludable and non-rival.

Necessity and intellectual curiosity are still the main drivers of open source development today. There are countless software projects that address a wide variety of needs, from printing utilities to computer games, from scientific tools to commercial applications. Thousands of volunteers from around the world contribute in these software projects, communicating problems and sharing ideas. Besides volunteers, one can observe more and more professional programmers who work with open source software. It appears that companies realize the power of tapping into the curiosity of the many.

To state that open source software has become important to firms and expect astonishment in 2010 is moot. Even the most ardent critic acknowledges by now the commercial potential of open source software. Examples such as the close collaboration between Microsoft and Novell or Oracle's acquisition of Sun, whose main products are all open source, are just the proverbial tip of the iceberg. Technology firms use and produce open source software with increasing ease and many to do so successfully.

A recent study suggests that 85% of surveyed firms use open source software (Gartner, 2008). The same survey asserts that usage is evenly divided between mission critical and non-critical processes. Industry experts estimate that companies are heavily involve in its production (Ghosh et al., 2002; Lakhani and Wolf, 2005). The estimated figures range between 38% to 54% of developers who are paid for their work.

In light of this situation, the question at hand is not anymore whether firms use open source software and contribute in its development. The predominant issues are rather how can they create profitable business models, why do firms contribute in open source software and what are the effects of corporate contributions on open source development. These are the questions we want to investigate in this thesis.

Our endeavor is worthwhile for another reason than the commercial importance of open source software. Corporate participation in open source software touches upon a number of different issues in economics and social sciences. It is, among others, a natural experiment for the private provision of a public good, for gift giving, for collective invention, for disruptive technologies, for technology outsourcing or for decentralized organizations. Little does it surprise then that academia strives to understand corporate involvement in open source software. This thesis focuses on three aspects within this vast field: the private provision of a public good, collective invention as well as outsourcing of technology.

How can firms provide a public good? This is a question that lies at the core of the economic analysis of public goods. Public goods are non-excludable and non-rival. This means on one hand that, once they are made available to someone, anyone has access to them. Nobody can be excluded. On the other hand, it also means that there is no scarcity of the product. The consumption of the good does not affect its availability to others. The nature of public goods render their private provision difficult. Economic theory predicts that firms and private individuals will generally underinvest in the production of the good.

We look at corporate open source software to understand the possibilities to provide a public good privately. Open source licenses create software that is, to some extent, non-exclusive. Anyone who uses the software can also modify and redistribute it. This renders applications with an open source license difficult to market. The firm needs to find alternative ways to create value for the consumer and sell their product or service. Reviewing the existing literature on corporate open source software, we point out common business models and possible forms of investment in open source software.

What are the effects of corporate interaction with open source communities? This question addresses collective invention and more generally open innovation. Open innovation proposes that the firm benefits from opening up research and development (Chesbrough, 2003). Involving external actors in research can lead to economies of scale or scope and knowledge spillovers. The notion that the firm benefits from collaborating in research and sharing information is not new. Muntz (1909), writing about the iron ore industry, already noted that

> "Each individual has some cherished bit of knowledge, some trade secret which he hoards carefully. Perhaps by sharing it with others, he might impart useful information; but by an open discussion and interchange he would, almost for certain, learn a dozen things in exchange for the one given away."

Corporate involvement in open source software provides us with useful information on the performance of one form of open innovation. To shed light on the effectiveness of involving corporate with voluntary developers and of knowledge spillovers, we estimate the coefficients of a production function of the average open source project. This estimation gives us an indication on how well open innovation works in open source software.

Why do firms contribute in open source software? This question is set into the broader context of technology outsourcing. We contend that corporate use of open source software is a kind of technology contracting. The firm signs the open source license in return for certain functionalities of the application. Contracting external suppliers can entail risks.

Transactions are costly to enforce, contracts cannot encompass all possible contingencies and specific investments might be made before transactions take place. These risks can deter the firm from entering into deals in the first place. To overcome these obstacles, the firm devises additional mechanisms to secure its transactions, e.g. vertical integration or long-term contracts.

The open source license is an incomplete contract. It does not encompass any specifications, deadlines or specific features neither does it allow for any future contigencies. We argue that these potential deficiencies in the contract between the firm and open source project lead the firm to dedicate programmers to work on the software. Corporate open source contribution is therefore a form of vertical integration.

For our empirical analyses, we use data on all open source projects which are hosted on the SourceForge website (Gao et al., 2007). SourceForge is an online repository that provides a platform for open source developers to manage their software projects and for users to find open source applications more easily. It is the largest of its kind with around 230,000 software projects and 2 million registered users (SourceForge, 2010). For comparison, Savannah, a similar repository hosts around 3,200 projects.

This database is extended with background information on registered developers. We relate the developers' email addresses with top-level domain names, download the corresponding website and check its content for several keywords.[1] Counting the frequency of occurence for these keywords, we can rank each website according to three categories: Academic, corporate or private. This method allows us to have a proxy of corporate involvement in SourceForge projects. We discuss the downsides of this method in Chapter 2.

This thesis is divided into three parts. Chapter 1 presents the literature review on corporate open source software. Chapter 2 focuses on the production of open source software and Chapter 3 deals with corporate contributions due to incomplete contracting.

[1]See Appendix A for the Python source code.

Chapter 1

Corporate Open Source - A Literature Review

1.1 Introduction

The improbable success of open source software stirs the interest of economists. For the last twenty years, open source software has gained more and more momentum in technology sectors. It is used for diverse applications such as servers, embedded systems or consumer products. Its success is improbable because open source software is provided voluntarily by a large number of private individuals, academics and, increasingly, by companies. What makes this phenomenon even more remarkable is the fact that open source software is a public good.

The private provision is a centerpiece in the economic analysis of public goods ever since the seminal works of Olson (1971) and Coase (1974). Traditional economic theory contends that public goods create market failures. A market failure arises because the private benefit derived from the good does not coincide with the social benefit. Consequently, the private provision of a public good is riddled with free-riding, coordination issues and conflicts of interests. Despite these obstacles, open source applications have managed to enter and, in some cases, even to dominate former monopolistic markets.

In this sense, the research on the corporate provision of open source software is an extension of the public goods literature. It sheds light on a particular case of a public good: Software with an open source license. Researchers illustrate various aspects of its provision. They uncover the incentives for individuals to contribute in its provision. Other scholars examine the mode of open source production and community governance.

A growing body of economic literature explores the reasons for corporate use and provision of open source and its implications for the software industry. Our objective in this chapter is to present a comprehensive survey of the economic literature on corporate open source software. Our approach is twofold. First, we look at why firms use open source software, in the course of which we shed light on the different open source business models. Then, we consider the reasons for corporate provision of open source software. Most of the papers in this field are empirical studies and there has been little theoretical work on corporate open source software. A notable exception is the literature on the strategic interactions between open source and proprietary software (Bitzer, 2004; Mustonen, 2005; Economides and Katsamakas, 2006).

The chapter begins with a discussion of the properties that make open source software a public good. We then turn to the reasons for corporate use of open source software. Next, we look at the different business models that have arisen around open source software and then consider the interaction between proprietary and open source software. The second part of the chapter deals with corporate contribution in open source software. We talk about the reasons for firms to contribute. Finally, we end the chapter with a discussion of the different corporate strategies for contribution which include the adoption of business processes, the protection of intellectual property and the types of corporate contribution.

1.2 Open Source as a Public Good

Open source licenses are the building blocks for open source applications. They lay out the rules for modification and redistribution. The aim of open source licenses is ultimately to open up computer applications, encourage collective development and promote redistribution (Rosen, 2004). To study and improve the software, developers need to access an application's source code, the programming instructions in human-readable form. To this end, the source code needs to be open, hence the term open source.

These licenses create a public good. Open source applications are non-rival and to some extent excludable. Because it is an information good, the open source application is by its very nature non-rival: One's own consumption does not diminish the availibility to others. A potential user of an open source application has to agree to the conditions of the open source license upon installing the software. Once he accepts the terms, he is a member of the community of this application, he benefits from the application, but at the same time has to abide by the rules of the license agreement. All such licenses delineate the way modifications to the software are treated and how one can redistribute the software. These restrictions make open source software in fact a specific kind of public good: A club good. The open source license acts as a gatekeeper who ensures that the rules of the open source community apply to every user of the software. In contrast to the textbook example, open source software does not suffer from congestion for all members. Open source communities may experience positive externalities through increased use. More users can help find programming glitches and problems, as Raymond (2001) states "given enough eyeballs, all bugs are shallow". This might not be true for contributors, though. As the number of contributors increases, congestion in development activities may occur.

A public good is non-exclusive which means that everyone can benefit for free from the effort of others. This creates a disincentive for each individual to provide the good in sufficient quantities because they may not extract the full benefits from their effort.

The lack of incentive results in underprovision or complete failure of the provision. This situation is the typical free rider problem. How is this relevant for open source software? Although open source communities are excludable, they are so only until one signs the open source license. Once a user is a member of the community, he has a non-exclusive access to the software and all its modifications. Following economic theory, we expect to find similar underprovision and lack of incentives in the open source community.

Moreovere, coordination costs can hinder the voluntary provision of open source software. The development of a software application imposes strong constraints on its organization. As Brooks (1978) notes in his famous phrase that "adding manpower to a late project makes it later ", traditional software production quickly runs into decreasing returns to scale. Brooks (1978) attributes rising coordination cost as the reason for decreasing productivity. In traditional software creation, changes in one part of the software have ramifications on all other parts of the software. Accordingly, each developer needs to know what all other developers do. As a consequence, the cost of communication increases exponentially as more developers join the project.

The ability to modify and redistribute open source software are important elements in its success as they may reduce the free-riding and coordination problem (Benkler, 2002). Modification is rendered easier through the modular design of large open source applications. It reduces the coordination costs implicit in organizing a large-scale software development project. Modular design breaks a complex system down into smaller parts, called modules. These modules can function independently as well as together to perform a more complex task with only a minimum of communication (Langois and Gazarelli, 2008). By hiding information, developers treat each module as a black-box (Baldwin and Clark, 2006): They do not need to know how it works only what goes in and what comes out. Each module has an interface with which programmers or other modules can interact. Standardized and open interface ensure that other developers can write code that attaches seamlessly to modules. Modular design hence avoids constant communication between development teams, they only need to define interfaces once and can then work independently. Modular software supports parallel development. Different modules can be developed at the same time without depending on the completion of other parts of an application.

Redistribution, on the other hand, allows individuals to share their personal modifications with others. The concept of innovations led by the needs of users finds many examples outside the software market: the industries for semiconductors, scientific instruments and windsurfing are but a few (von Hippel, 1988; Shah, 2000). Its main tenet is that users are in a better position to solve their problems than producers. This is so, because some information is *sticky*: User requirements and preferences are either too specific or too costly to transfer from the users to the producers. This means there is an information asymmetry between the two groups as well as an incentive for users to customize their product. Users therefore begin to add small improvements to the product themselves. The Internet facilitates sharing improvements and exchanging information about problems, especially for software. It is a cheap distribution and communication channel. Lakhani and Wolf (2005) and von Hippel (2001) show that user needs are important motives for individual contributions. These user needs ensure that developers have a personal interest in contributing code and adding new features. Users become developers in open source packages. This personal benefit reduces the risk of free-riding. The private benefit of adding a feature may exceed the cost of writing it (Bessen, 2005).

1.3 Corporate Use of Open Source Software

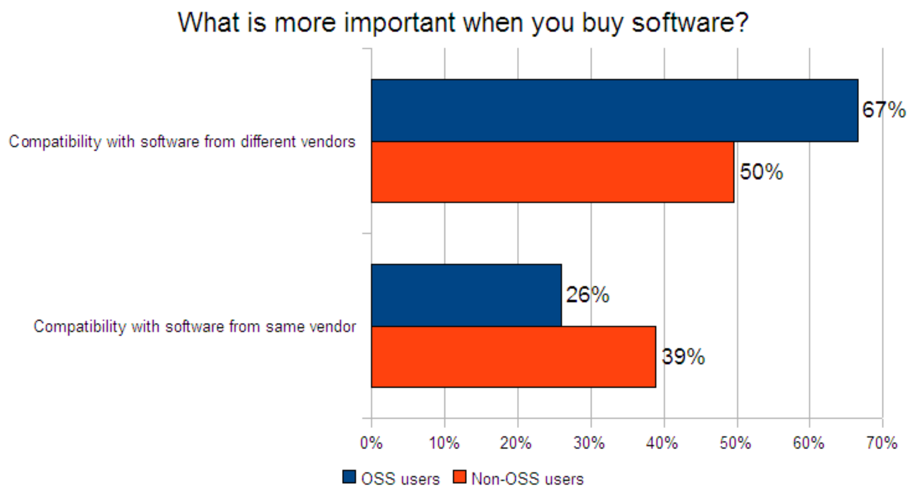
Surveys show that firms predominantly use open source software for their servers (Bonaccorsi and Rossi, 2003b; UNU-MERIT, 2006). Scholars find two main reasons for corporate use of open source software (Wichmann, 2002b; UNU-MERIT, 2006): Cost reduction and standardization.

Companies use open source software to reduce cost. As soon as the quality of open source software becomes good enough for normal use, it can be a less expensive alternative to proprietary software (Hawkins, 2004). Wichmann (2002b) investigates what managers consider a reason to use open source software. He finds that 56% of respondents agree that the pricing and licensing policy is the main motive. The cost reduction does not simply arise because there are no *ad hoc* licensing fees, however. Royalties are only part of the total cost of ownership, which include also training and maintenance (Weiss, 2005). Hence the actual decision to implement corporate software depends also on the necessity to customize the software during implementation as well as training and maintenance services.

A lock-in arises because switching a software, once it is put in place, is costly. These cost can come from the investment in human skills and training as well as from compatibility issues. On the one hand, the use of a software requires a certain level of experience. Depending on the specificity of the software, replacing it can require additional expenditures for staff training. On the other hand, computer software is rarely used by one person alone. Documents and files are exchanged regularly among different users. Switching software may inhibit the exchange or, in the worst case, render a user unable to interact with other users all together. A firm can thus get locked-in on one software application.

The cost advantage of using open source software arises when firms need to substantially customize the software. Firms, like private users, can adapt open source software to their specific needs (von Hippel, 2001). This flexibility allows a firm to adjust to change more quickly. This requires corporate developers to have the technical expertise to write and adapt the open source software. In case the developers lack the necessary human capital, a firm can rely on external open source service providers. These services are more competitive than those for proprietary software.

Standardization is another reason to use open source software. Standards are important in an industry in which multiple different applications and systems regularly inter-operate. West (2003) sees one of the main advantages of open source software in facilitating open platforms and infrastructures. Platforms and standards are costly to establish and open source software offers another means to coordinate standards among



Source: UNU-MERIT (2006)

Figure 1.1: Importance of Interoperability

firms. The open source licenses provide a common framework which ensures that the platform remains accessible to everyone. The survey by UNU-MERIT (2006) sheds light on the importance of open source software as an open platform. It finds that open source users place a higher value on interoperability than users who do not use open source software, see Figure 1.1.

Wichmann (2002a) corroborates these findings. In his study, 33% of respondents attach at least a high importance to open source software for the firm's information technology infrastructure, which include applications to support ongoing business processes and communications.

The use of open source software entails a risk also (Jeon and Petriconi, 2009). The lack of warranties or a corporate supplier can be a burden when liability issues arise. Firms which accept open source licenses waive all warranty claims against open source project or its developers. This means that, in case the software malfunctions, a firm has no possibility to obtain reimbursements or charges for breach of contract from the development community. At the same time, the lack of protection creates a business opportunity for specialized software firms to provide these warranties.

1.4 Open Source Business Models

Since open source software is a public good, companies cannot simply sell the software. They need to bundle the open source software with a marketable product or service. Business models based on open source software center around the complementarity between the open source application and a service or product. This complementarity can take two forms. Either there is a complementarity in demand or a complementarity in production. Complementarity in demand is a positive network effect and means that as open source software is adopted more widely, the demand for the marketable product increases too. Hence, as the demand for the open source product increases, the firm can sell more services or secondary products. Complementarity in production on the other hand stands for a positive network externality and means that as the developer base for the open source software increases, a firm receives more and more bug fixes, program improvements and support. These can then be included in other marketable products based on the open source software. Doing so reduces the cost of production and entails external innovation (Chesbrough, 2003).

Along these lines of complementarity of demand or production, economists distinguish four broad open source business models. Firms can (i) provide services based on the open source software, (ii) supply products which make use of open source software, (iii) offer modules and add-ons to open source software or (iv) sell corporate, closed source versions of the open source applications.

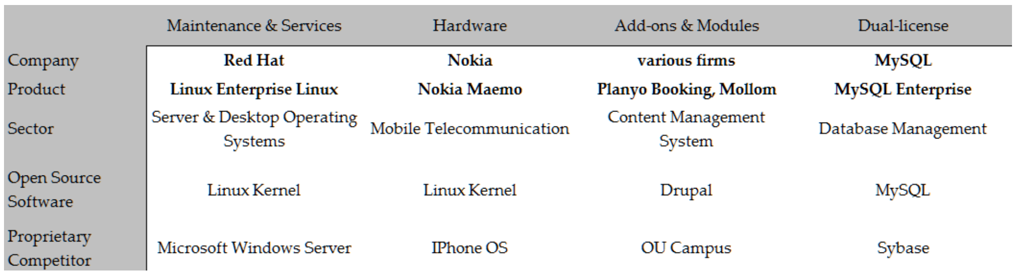
Service providers for open source software offer the open source software for a small fee and sell support, maintenance and other services. The open source software drives the demand for the complementary services. The higher the installed base of the open source application is, the higher the demand for additional services will be. The firm's competitive advantage is its human capital (Hecker, 1999). Because of its strong involvement in the development process, the firm is in a unique position to offer superior services and warranties for the open source software. Due to the nature of open source software however, complementary service providers can face fierce competition (Behlendorf, 1999). Since the software is open, rivals can easily enter the market, adopt the application and provide similar services.

Hardware producers offer their product in conjunction with open source software. There is a complementarity in production because firms can outsource some of their development effort to external developers, either private or corporate, for free. Hence the open source software reduces their production cost - in addition to saving the licensing fees. Unlike service providers, hardware producers face less competition from their open source engagement. The specificity of the hardware protects them from imitators, while most external developers either work in different markets or are hobbyists (Henkel, 2006). Customized hardware is of little value to other producers with different hardware specifications. This specificity reduces the risk of free-riding from competitors as they cannot adapt the open source code for their products.

The third business model is the sale of complementary modules or add-ons. Open source software is often build in a modular way. Using this modular architecture, firms can market specific modules to existing applications (Behlendorf, 1999). These add-ons fit into the main software and provide additional services or features for paying customers. Customization and adaptation to the needs of clients are the main sources of revenue. Here, a firm benefits from complementarity in demand. The link between the open source application and the module implies that as the application becomes more popular and its user base increases, the demand for the firm's modules increases too. The reliance on the main open source software however also entails a downside. The firm depends on the success of the overall software and the decisions taken by the development community. Involvement in the development and the decision-making process of the main software project is one means for the firm to protect its open source investment.

The dual-license business model is closest to the traditional software vendor (Välimäki, 2003). A firm provides an open source and a close source version of the same program. Both versions are basically identical apart from the license. Customers can obtain a

Table 1.1: Open Source Business Models



	Maintenance & Services	Hardware	Add-ons & Modules	Dual-license
Company	**Red Hat**	**Nokia**	**various firms**	**MySQL**
Product	**Linux Enterprise Linux**	**Nokia Maemo**	**Planyo Booking, Mollom**	**MySQL Enterprise**
Sector	Server & Desktop Operating Systems	Mobile Telecommunication	Content Management System	Database Management
Open Source Software	Linux Kernel	Linux Kernel	Drupal	MySQL
Proprietary Competitor	Microsoft Windows Server	IPhone OS	OU Campus	Sybase

proprietary license and thus keep their modifications hidden from the open source community. This way they can create value-adding features and customizations to the application without sacrificing their competitive advantage. This business model offers the best of both worlds (Välimäki, 2003): it harnesses the development effort of an open source community to improve the software, while, at the same time, it generates revenue the traditional way through selling software licenses.

1.5 Open Source and Proprietary Software

As software firms adopt open source business models, the rivalry between open source firms and proprietary software producers becomes a prevalent topic and one with high stakes for both parties concerned. Usually the incumbents in these markets follow a traditional proprietary business model and so they face unlikely opponents: companies using communities of volunteers. Outsourcing software development to a group of people who work for free gives open source firms a considerable cost advantage towards proprietary firms. Empirical evidence suggests that, until now, open source firms are far from dethroning the proprietary incumbents (Bonaccorsi and Rossi, 2003b). But this coexistence of the two different business models is all the more reason to search for factors affecting this type of market outcome and to shed light on the underlying dynamics of the market and firm interaction.

Because of the ease with which one can simplify this problem to a duopolistic market structure, it lends itself to game theoretic analysis. A variety of theoretical economists explore the duopolistic competition between proprietary and open source software providers (Bitzer, 2004; Mustonen, 2005; Casadesus-Masanell and Ghemawat, 2006; Economides and Katsamakas, 2006). Their works differ with respect to strategic interactions, the firms' cost structures and the time horizon of the analysis.

Bitzer (2004) considers a market in which a proprietary software firm competes with one open source firm. Their two pieces of software are sufficiently differentiated that both firms can charge a price higher than marginal cost. The open source firm does not invest in innovation. By contrast, the proprietary firm spends money on innovation, e.g. to add functionality or to increase compatibility. Bitzer (2004) is able to show that in this setting, the proprietary firm coexists with an open source rival in the long run, if the differentiation between their two products remains large enough due to innovation on the part of the proprietary firm. This finding entails two corrolaries. First, a proprietary firm has an interest in differentiating away from the open source application, because doing so secures a market segment and a client base. Second, differentiation however is costly and depends on the extent to which the proprietary firm funds innovation. If the cost of innovation becomes too high, differentiation decreases and the proprietary firm's profits may fall below average cost, leading it to exit the market.

Mustonen (2005) looks at the factors that lead proprietary firms to render their software compatible to competing open source software. The two computer applications, one proprietary and the other one open source, are differentiated products and create positive network externalities to the respective consumers. By increasing the user base, compatibility is a two-edged sword. On the one hand, it benefits the proprietary firm by increasing the network effects for the firm's own consumers. The consumers benefit from services offered on the open source application and thus also value more highly the proprietary software. On the other hand, the proprietary company may lose its competitive advantage over the open source software, if it has itself the larger user base to begin with. The firm's choice depends on the strength of the network effect and the quality of the open source application. Mustonen (2005) finds that, as the size of the network effects increases, the firm's incentives turn in favor of compatibility. By contrast, a proprietary incumbent might find it profitable to remain incompatible to ensure his price setting power, especially if the two software products are close substitutes.

Casadesus-Masanell and Ghemawat (2006) extend the literature with a dynamic analysis. They model a duopolistic market in which demand-side learning effects create network externalities over time. As consumers adopt the open source software, later consumers learn from these previous decisions. These learning effects trigger a momentum in favor of the open source software. Casadesus-Masanell and Ghemawat (2006) assert that, regardless of the strength of the learning effect, the proprietary application remains in the market as long as the quality difference between the two pieces of software is large enough. Not surprisingly, the arrival of the open source alternative reduces the profit of the proprietary incumbent and in general increases total welfare.

Economides and Katsamakas (2006) use a two-sided market analysis to investigate the impact of an open source alternative on a proprietary software producer. A two-sided market is essentially a platform which trades with several distinct groups. These groups benefit from each other's presence on the platform, which translates into network externalities across the platform. The platform in turn can internalize some of these network benefits and sets its prices for the different groups accordingly. This kind of market is particularly useful to analyze software platforms, such as operating systems where there are consumers on one side and application providers on the other.

Economides and Katsamakas (2006) derive three conclusions. First, they show that, when consumers value highly variety in applications, application providers earn higher profits on the proprietary platform than on the open source one. This result stems from the inability of the open source platform to internalize the network benefit accrueing from consumers. Second, the authors show that with free entry, open source platforms have a larger variety of applications than proprietary platforms. This finding is due to the zero price of open source platforms and the resulting adoption among consumers. Third, Economides and Katsamakas (2006) also contend that an industry-wide adoption of an open source platform is welfare enhancing due to the larger availibility of complementary applications. In all, they are able to formulate a market setting which can explain why proprietary firms can coexist with open source firms.

1.6 Contribution

Corporate open source contributions are investments in software communities. As with other investments, a firm wants to reap the returns of its open source engagement. We have seen in the last section that a firm cannot simply market the resulting product. Alternative benefits have to be realized. The reasons to contribute depend on corporate strategy and the business model of a firm. Bearing this in mind, we find that there are either technical or strategic reasons to contribute (Wichmann, 2002b; Henkel, 2006; Osterloh and Rota, 2007). Among the technical reasons are complexity, compatibility and standardization. Strategic reasons comprise gaining a reputation within the open source community and competitive considerations.

Complexity can be a reason for firms to contribute in open source development (McGovan, 2001; Bessen, 2005). Bessen (2005) contends that software applications consist of a large number of features and that, in contrast to proprietary pre-packaged software, open source software allows customization to individual needs. Proprietary software producers offer pre-packaged applications that address the average consumer. Due to transaction cost and asymmetric information, he cannot provide every feature for every customer. Firms have heterogeneous preferences for functionality and some might benefit from customizing the software on their own. Contributing in open source development therefore allows these firms to modify an existing software and more appropriately match their needs with the application.

Compatibility is another reason for corporate contributions. Firms contribute to enable compatibility between their product and the open source software. These contributions are generally rather limited (Bonaccorsi and Rossi, 2003b) and minimize the exposure of the corporate product to the open source software - and restrictive open source licenses (Wichmann, 2002b).

Standards are pivotal elements in software platforms. As these platforms transform from valuable, differentiating assets into low-cost components in various hardware products, firms are interested in reducing the cost of maintaining them. One such way is through open source software, which remains accessible to everyone. The development and maintenance cost are divided among all participants. Maintaining the open source platform thus becomes a collaborative effort of private individuals and corporate developers from various firms. Theoretically, this could lead to an arms race in contributions. Companies want common standards for open source platforms. Moreover, they prefer to have *their* standards adopted in these platforms, hence firms may intensify their open source engagement to get their standards integrated in the project.

Showing good open source citizenship is one more reason for corporate open source activity (Osterloh and Rota, 2007). Open source developers generally follow a kind of hacker codex (Raymond, 2001). This means that there are certain social norms and customs which determine how developers contribute to the development and communicate with other developers. By contributing, a firm wants to show that it plays by the rules. The collaboration between community and firm depends on trust. Sharing code with the community earns trust and creates good will. This in turn leads to a more beneficial cooperation between firm and open source community.

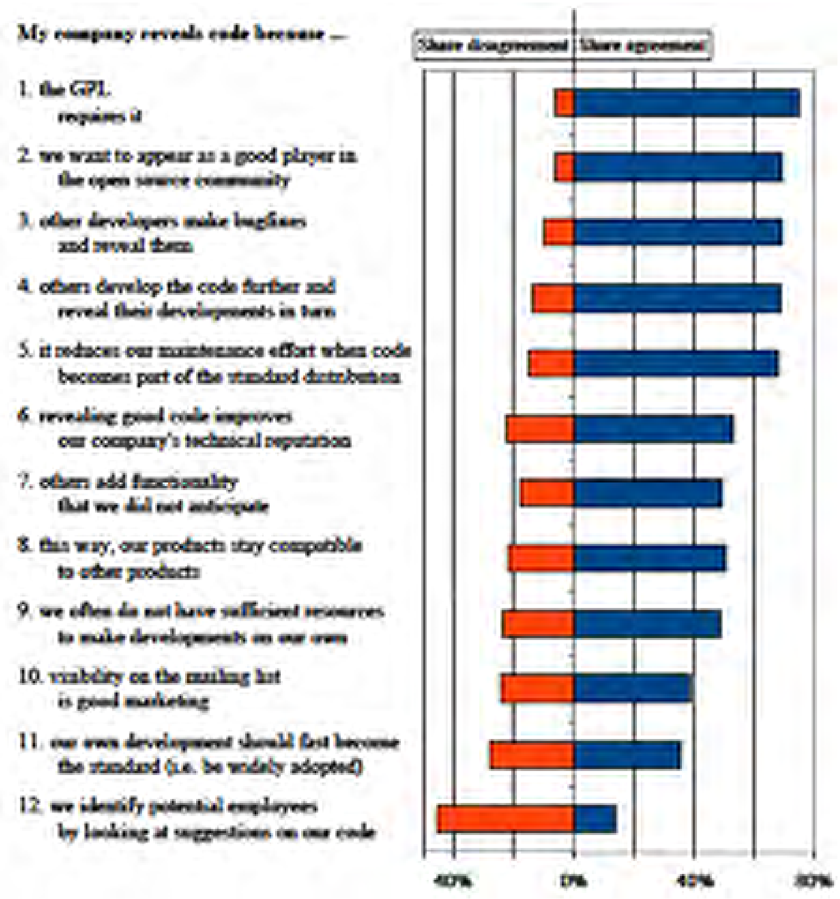
Trust between community and firm is especially important when it comes to corporate-led or corporate-owned open source projects (Shah, 2006). The difference between leading and owning a software project lies in who keeps the copyright of the source code. A corporate developer might act as the project administrator, i.e. he heads the project, but the copyright for the source code still remains with either the each individual contributor or the development community. By contrast, a company-owned open source project owns the source code and obliges its contributors to transfer the copyright to the company.

Corporate leadership and ownership influence the effectiveness of open source development (O'Mahony and West, 2005; Shah, 2006). More and more companies use and initiate open source software as useful additions to their product lines (Bonaccorsi and Rossi, 2003b; UNU-MERIT, 2006). Developers are suspicious of corporate projects, however. They are wary of free-riding on the part of the firm as well as a lack of attribution of their contributions (Shah, 2006). These fears stem from the perception that firms are less bound by social norms or reciprocity and more bound by profitability. Yet, developers contribute because either there is no private open source alternative or the corporate project offers better quality.

In her study, Shah (2006) indeed finds that there is a large difference in developer motivations between private and corporate open source projects. Respondents disapprove with company-run projects because the company alone controls the copyright of the project and therefore is the sole owner of the source code. Contributors suspect that the firm can arbitrarily restrict the use and distribution of private contributions. As a consequence, individuals feel less obligated to reciprocate with the development community.

To overcome these reservations, companies have to interact carefully with open source communities. They can create goodwill within the open source community through several measures (Shah, 2006; Osterloh and Rota, 2007): (1) involving private contributors in the decision-making process, (2) using a restrictive open source license which effectively prevents corporate code hijacking, or (3) hiring renowned open source developers to show the commitment to the open source idea.

Competitive considerations can drive open source contributions as well. In most fields, open source projects compete with proprietary applications. Firms can use open source software to undermine a competitor's dominant position and commoditize an application (Ven and Mannaert, 2007). They contribute to the projects to improve it up to the point where the program is good enough for day-to-day use. Undermining an incumbent's market position entails two benefits. Firstly, it reduces the cost for upstream markets. Firms which use proprietary platforms may want to replace them with open source alternatives to cut the licensing cost. Secondly, firms may benefit from the release of proprietary



Source: Henkel (2006)

Figure 1.2: Reasons to Contribute

software by an increased demand for complementary products.

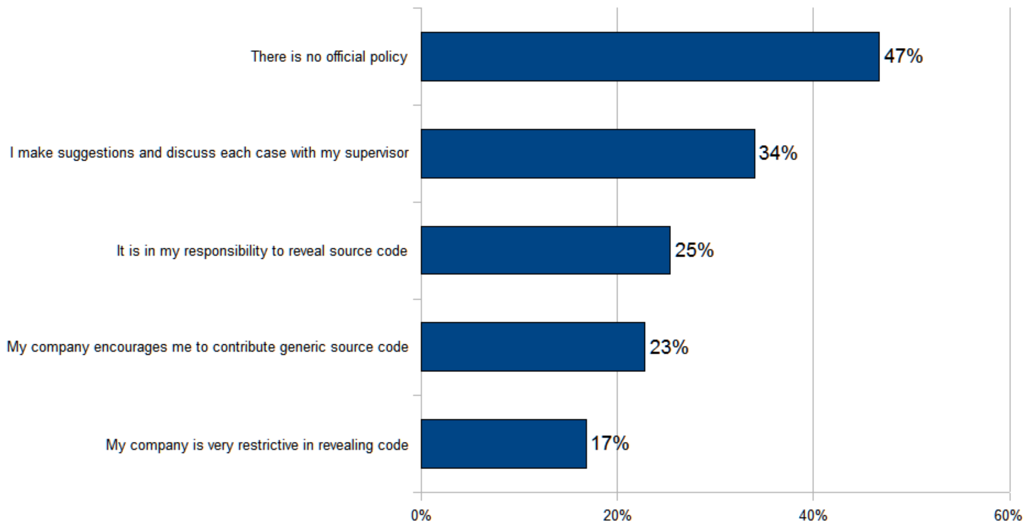
Henkel (2006) gives an overview of the corporate developers' opinions on why their firm reveals source code, see Figure 1.2. He finds supporting evidence for most of the aforementioned motives. Most developers agree that they contribute because the open source license requires them to do so. This response shows that firms, unlike individual contributors, are less driven by ideological considerations - the license makes them reveal source code. Obtaining the trust of the open source community ranks high. What is more, responses addressing reciprocity or cost arguments rank high too. These responses are related to the a firm's good will in the community. Individuals who trust a company are more inclined to provide bug fixes and new features. Hence the more a firm contributes the more individuals trust the company; the more they trust the company, the more they contribute. Firms realizing this link have an incentive to contribute to open source projects.

1.7 Corporate Strategies

Research suggests that, before a firm can release source code or contribute in open source projects, it is advantageous to create specific open source strategies (Hecker, 1999). Open source contribution entails software specific challenges. To reap the full benefit of the open source engagement and protect a firm's valuable assets at the same time, a firm could consider several aspects. It can delineate rules that define which pieces of code could be revealed (Ven and Mannaert, 2007; Henkel, 2008) and can establish a platform for open source software within as well as outside the firm (Hecker, 1999; von Hippel and Katz, 2002).

Corporate developers who work on open source projects face a dilemma (Henkel, 2008). Their affiliation with the open source community may conflict with the loyalty to their firms. Because they act as interface between the company and the open source project, they have a pivotal role in the open source engagement of the firm. The decision to reveal code to the community ultimately lies with them. Rules may strike a balance between the uncertainty faced by the developer on what to disclose and managers' risk perception of their open source engagement. Henkel (2008) proposes that these rules could be drawn up in collaboration with management, intellectual property lawyers and the developer team. Figure 1.3 shows the share of corporate developers who agree with the given statements.

Additionally, a firm can create internal processes for open source software development (Hecker, 1999). These processes include the preparation of the source code as well as a change in the mode of development. Source code for in-house use may differ from source



Source: Henkel (2008)

Figure 1.3: Corporate Policies towards Revealing

code published for an open source project. The system architecture might be adopted for release and the application modularized to accommodate the open source mode of production. This might also make it easier to decide which parts should be disclosed and which should not. Moreover, firms often use proprietary third-party technologies in their applications. Before release to the open source project, the firm can either seek permission to use proprietary third-party technologies or replace it with an open source compatible pieces of code.

The inclusion of potentially large numbers of external developers is an advantage of open source software. Apart from providing the source code, a firm can create a positive social context to motivate and coordinate open source developers. Dedicated corporate coordinators with experience in open source projects can impose stronger control on the open source community (Eilhard, 2008a). Henkel (2008) points out, however, that corporate coordinators and developers should not have a strong ideological attitude towards open source software as this will lead to excessive revealing of corporate intellectual property.

In case a firm initiates an open source project, it can also provide proper external facilities, for instance user forums, tools to report bugs and to submit contributions. Moreover, firms may also provide user-tool kits (von Hippel and Katz, 2002). These software development kits not only help to facilitate the adoption of the project, but also give the firm innovations as well as information about user needs that can be used for future products (Jeppesen and Frederiksen, 2006). These kits can speed up development considerably (von Hippel and Katz, 2002). Moreover, the external infrastructure creates a good reputation for the firm (Dahlander and Magnusson, 2005). Reputation gives a firm a stronger clout on open source project leadership.

1.7.1 Protection of Intellectual Property

Involuntarily releasing valuable ideas is an risk in corporate open source engagements. In fact, the mode of open source production consists of regularly disclosing information to everyone. Information is particularly prone to misappropriation and imitation, because it is a public good. Once disclosed, anyone can use the knowledge in the information and the commercial value for the inventor is severely reduced. Traditional means to protect intellectual property are patents, copyright, lead time, secrecy or complementary marketing or manufacturing (Cohen et al., 2000). These measures are often used in combination to provide stronger protection. Apart from traditional means, the literature finds means specifically designed for open source software.

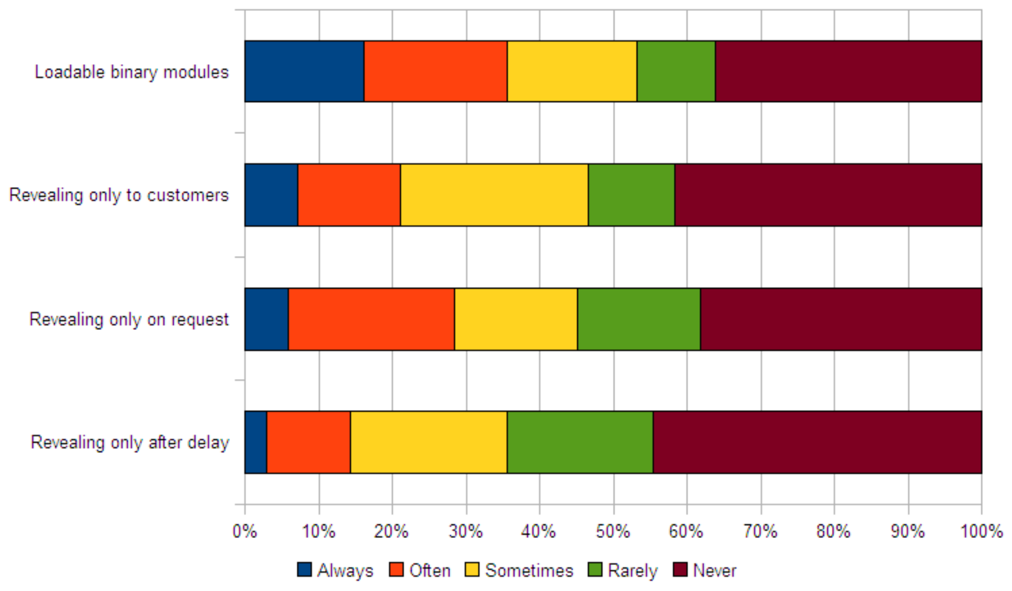
Research identifies several ways to protect intellectual property specific to open source software (Bonaccorsi and Rossi, 2003b; Henkel, 2006): Selective revealing, hybrid licensing schemes, delaying disclosure and circumventing the licenses. A firm does not need to reveal all the modifications to the open source community. Even though restrictive open source licenses require the disclosure of all modifications, a firm does not have to actively publish its source code. The licenses only ask for the availability of the source code, hence firms may either distribute the source code on demand or, when the number of users is very small, arrive at the common understanding with the users to not publish the source code at all (Henkel, 2006).

Another means to protect corporate intellectual property is to use a mixed licensing scheme (Bonaccorsi and Rossi, 2003b; Rosen, 2001). Restrictive open source licenses do not place a limitation on the ability to distribute an application under proprietary licenses. A firm can thus distribute an open source version and a closed source version of the same software. Users of the proprietary version can keep their modifications secret and still benefit from a large external development community.

Delaying the publication of the source code is yet another way to secure one's own intellectual property (Henkel, 2006). Lead time is especially important in software and computer markets. By extending the time between product development and its market launch, a firm delays the source code disclosure and effectively fend off competitors. This time advantage then secures its investment in the development of the software. Henkel (2006) notes this behavior in the embedded software case.

Finally, a firm can circumvent restrictive licensing conditions by hiding open source code within proprietary packages (Ven and Mannaert, 2007) or by simply not providing active support for the source code (Henkel, 2006). The former method, although technically not allowed under restrictive open source licenses, is rather effective. Once the source code is incorporated in a closed source module, identification of the infringing parts of source code is difficult. The latter method discourages development by individual developers. A firm might not provide clear documentation of its software for instance. These practices - if revealed - are proscribed by the open source community. Henkel (2006) offers a measure of how often firms rely on these means (Figure 1.4).

In addition, Dahlander (2005) explores the use of traditional intellectual property protection in five open source software firms. He finds that firms use a mix of protective measures for their product. Four out of five firms rely on copyright to protect their contributions, followed by secrecy, three out of five. Only two out of five firms protect their intellectual property with first-mover advantages or complementary assets. **?** explore the relationship between a firm's patent and trademark portfolio and its open source



Source: Henkel (2006)

Figure 1.4: Protective Measures

engagements in their study of 461 firms. They find that firms with large portfolios of software patents or hardware trademarks are more likely to release open source software. This could indicate that firms can use open source software as a complement to their patented products. In contrast, they also reveal that firms with a large number of software trademarks or hardware patents are less likely to release open source software.

Proponents of open source software often depict software patents as its nemesis (Perens, 2006). In their view, the existence of patentable software code is the greatest threat to the open source development process. Evans and Layne-Farrar (2004) point out the main lines of the conflict between open source software and software patents supporters.

Restrictive open source licenses require mandatory licensing of software patents. To accord with the software license, any patented code has to be licensed to all users of the open source software for free. This one-size-fits-all condition does not pose a problem for one's own patents, but it does for third-party patents. The open source license basically voids any means to recoup cost for research and development. Less-restrictive licenses, on the other hand, allow proprietary software patents to be merged with open source software.

Among the different reasons against software patents, the most important ones are the reduction of the *software commons* and the weakness of software patents (Perens, 1999). Open source proponents argue that open source software creates a software commons, a stock of knowledge shared by all developers and accumulated through the code contributions in open source projects. Software patents, to use the image of pasture, creates a fence around parts of this commons and excludes the public. Put differently, software patents limit the possibility to freely include all available source code in open source software. Evans and Layne-Farrar (2004) assert that this view only considers the static perspective, whereas the dynamic innovative process is neglected. Were software patents to promote innovation, their elimination would decrease the incentive to create new software and thus less software would be written.

Bessen and Maskin (2000) counter this argument. They contend that, if innovations are sequential and complementary, patent protection can inhibit their invention. Sequential innovations build on preceding inventions, such as early versions of Microsoft Windows were build on DOS. According to Bessen and Maskin (2000), complementary innovations are different research approaches that lead to the same goal. Because of the complementarity, the overall likelihood of invention increases with each approach. There are many examples for complementary innovations in open source applications: Gnome and KDE, MySQL and PostgreSQL or Firefox and Epiphany. Consequently, a firm obtaining first a software patent can prevent its competitors from developing further innovations based on similar ideas. In case the patent-holding firm lacks the creativity of its competitors, patents slow down the rate of invention.

The weakness of their validity is another often-cited downside of software patents. Patents are supposed to protect inventions which are non-obvious, new and useful. The weakness of software patents stems from the filing of oftentimes old or obvious inventions. Weak patents are *per se* not a problem - litigation can ultimately determine the validity and strength of a patent. The real damage arises from injunctions, which bare all sale of potentially infringing products (Lévêque and Ménière, 2006). Faced with an immediate loss in revenue and the prospect of a long litigation process, firms are willing to pay high royalties even for weak patents. Evans and Layne-Farrar (2004) contend that weak patents will disappear with time as patent offices begin to gather a reference database on prior art in software patents and can then better determine the novelty and non-obviousness of the patent claims.

1.7.2 Types of Corporate Contribution

The extent to which a firm interacts with a community depends on its business model. In case a firm relies heavily on external development, it needs to interact more closely with the project. On the other hand, a firm may simply scan communities and projects to keep up-to-date on software development. In this case, a firm free-rides on the development effort of the open source project without contributing back to the community. The decision to contribute depends on the trade-off between the benefits and the cost of the open source engagement.

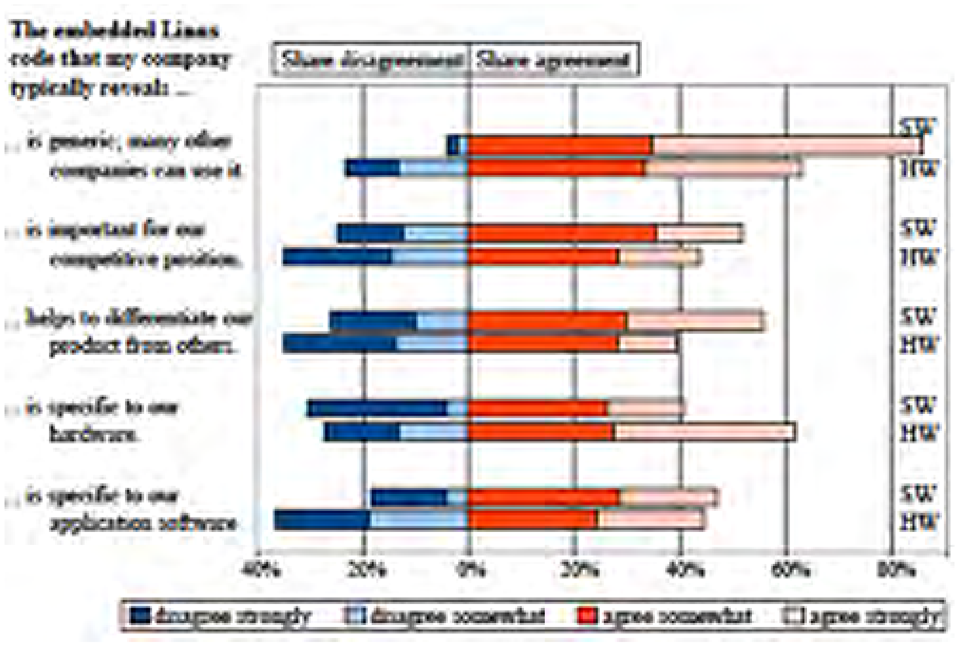
Along these two cases, Dahlander and Magnusson (2005) find three different approaches to corporate involvement in open source projects: symbiotic, commensalistic and parasitic. The *symbiotic approach* describes a situation in which the firm actively interacts with and supports the open source development community. The *parasitic approach* is the exact opposite. Here, a firm focuses on its own benefit regardless of the effects to the open source community. The *commensalistic approach* is a compromise in which a firm co-exists with the open source community. Although it benefits from developments provided by the community, it interacts and gives back little.

Using the symbiotic approach, a firm considers the open source development community as a direct extension of its knowledge base. It interacts regularly and takes an active interest in the leadership of the open source project. Dahlander and Wallin (2006) explore the intensity of communications within an open source project. They find that firm-sponsored individuals communicate more with others than vice versa. Additionally, these corporate participants interact with core members of communities. There are differences between firms which are dedicated OSS companies and firms which are incumbents in the software market. Henkel (2006) investigates the revealing behavior of corporate developers in a embedded Linux projects. He finds that firms reveal almost half of their contributions to the community. These results vary significantly among firms, but not across industries. The revealing behavior depends on firm characteristics, such as size and experience with open source software. Positive experience increases corporate developers willingness to contribute.

Figure 1.5 represents the types of corporate contributions revealed to open source projects. The results are split between hardware (HW) and software producers (SW). One can see that the surveyed firms mostly disclose generic pieces of code. Interestingly, the share of firms that release critical source code is high. One possible explanation is that software firms use code releases to increase their reputation and to signal technical excellence. Another explanation may be excessive revealing on the part of corporate developers who, lacking clear rules for revealing, contribute too much in open source projects.

In a recent study, Lamastra (2009) sheds light on the impact of corporate contributors on the quality of open source development. She finds that the quality of development activities increases in terms of code contributions. By contrast, all non-development activities of a community decrease due to the involvement of corporate developers. Lamastra (2009) suggests that this indicates a short-term perspective on the part of the firm and may lead to higher future cost.

The parasitic approach is one of pure free-riding. Firms copy and use source code provided by software projects without reciprocating. What distinguishes it to the commensalistic approach is the negative effect it has on the development community. The community may perceive a free-riding firm as such, if it violates rudimentary social norms and principles. Since this approach is not a sustainable way to interact with open source



Source: Henkel (2006)

Figure 1.5: Types of Corporate Contribution

projects, few firms would claim to be parasitic. Yet, the line between what constitutes simply passive use of open source software and intentional free-riding and abuse of the open source development effort may be subjective and depends on a community's perception of acceptable behavior. Bonaccorsi and Rossi (2003a) find that 46% of surveyed firms selling open source related products or services do not actively contribute in open source projects. Henkel (2006) obtains similar results in his sample: 51% of firms do not reveal their contributions to the community.

The commensalistic approach describes firms which co-exist with open source communities. Even though they benefit from the communities their own direct contributions are few. In contrast to pure free-riding, they honor the social norms and try to gain acceptance in the open source projects. In case they reveal code this code generally is generic. Bonaccorsi and Rossi (2003a) identify a lot of firms who only contribute in a few projects. 68% of the surveyed firms are active in at most 2 projects and only 8% in more than 10. They then relate social, economic and technological motives to firm activity in projects. They find that firms with high economic motivations coordinate less projects. Also firms which are more socially motivated do not contribute in more projects. In contrast, the number of accepted contributions increases with social motivation. Moreover, Henkel (2008) finds that the level of identification with the open source community influences the level of contributions in an open source project. He shows that corporate developers, although they identify with the community, feel a less strong commitment than private developers.

1.8 Conclusion

In the past ten years, economic research has come a long way in understanding why firms use open source applications and why they take part in their development. This chapter showed that open source software and commercial goals are compatible and that indeed there is an economic rationale to develop open source software.

The literature gives two main reasons for corporate use of open source software. First, open source software can give cost advantages. Apart from the lack of licensing fees, open source applications also entail lower maintenance cost and are less prone to lock-in. Empirical studies support this contention. They find that companies use open source software to avoid these cost. Second, open source software can promote standardization. Establishing a common technology platform is costly and difficult to coordinate across different parties. Open source software offers a common framework for collaboration and rules to coordinate collective development.

We present five motives for firms to contribute in open source software. Economists contend that firms contribute in open source software because computer applications contain a large number of features. The inability of the proprietary software producer to provide all features to every client induces firms to contribute in open source software and produce these feature themselves. Enabling compatibility can be another reason. Firms contribute to render their product compatible with an open source application. The literature also brings forward that companies participate in open source development to establish a standard. The control over standards and platforms is pivotal in the software sector. The more a firm contributes, the more it influences the creation of a potential standard. Companies also contribute to gain reputation in open source communities. Voluntary contributors are wary of corporate involvement in open source development. To counter the suspicion of corporate free-riding and opportunism, firms contribute and show their adherence to the open source ideology. Companies can also contribute for strategic reasons. Software markets often have a dominant incumbent who is protected by high barriers to entry. Open source applications offer a possibility to undermine the position of the incumbent. Contributing in open source software thus is a way to put pressure on the incumbent and to changes the software sector into a contestible market.

Our findings on corporate provision beg the question whether we need government aid to promote the adoption and development of open source software. Some governments put in place requirements to choose open source software over proprietary software in public institutions. Germany, for instance, has several initiatives to oblige the public sector to adopt open source software. In addition, there are several programs to subsidize the production of open source software. The United States subsidizes firms that create open source software, for example the Beowulf Linux Clusters. Is there a market failure that might justify governmental action?

There are two possible sources of market failure with regards to the software sector. Due to network effects or learning effects, consumers face high switching cost: They can only change the piece of software by incurring a considerable cost in terms of learning the new software or incompatibilities with users of other computer applications. They are locked-in on a piece of software which in turn might lead to a monopoly in the specific software market. Another source of market failure is the existence of knowledge externalities. These knowledge spillovers can accrue as a result of using or producing the software. Users share their knowledge about the software. Developers exchange information on specific programming issues. It is important to note that these spillovers might benefit the firm also. In case these knowledge externalities are higher for open source than for proprietary software, the government might increase social welfare by promoting open source software. So, is there a case for government support of open source software?

Leaving aside political economy, our discussion of open source adoption and business models indicates that open source software is quite capable of competing with proprietary software without state intervention. It is safe to say that lock-in effects exist in some software sectors and create inertia in software adoption, but we also see that technologies are highly dynamic and, because of this, render software markets strongly contestible. A firm that enjoys a monopoly in one niche, may find itself in fierce competition as technologies change. Governmental action, however well intended, might come too late or provide the wrong incentives. Knowledge externalities, on the other hand, are not limited to open source software. Proprietary software firms also establish means to share information and foster knowledge sharing among users and developers. There is as yet no empirical evidence that suggests that these knowledge spillovers are lower than the ones in open source software.[1]

In all, we see that the analysis of corporate provision of open source software is a worthwhile enterprise. What is more, the insights of our discussion are valid beyond the software sector. The challenges posed by collectively-provided goods as well as by digital distribution channels will not go away and are likely to increase in the future.

[1]To read more on governmental subsidies and open source see (Schmidt and Schnitzer, 2003)

Chapter 2

A Look Inside the Forge: Developer productivity and spillovers in SourceForge projects

2.1 Introduction

“The ability of the [open source software] process to collect and harness the collective IQ of thousands of individuals across the Internet is simply amazing.” This comment stems from a Microsoft software engineer made in the so-called Halloween memos in 1998 (Valloppillil, 1998). Already ten years ago, the world’s largest software producer understood the potential of open source software production. Today, this original productive model has demonstrated economic success. As a matter of fact, major proprietary applications face serious competition from open source rivals such as Linux or Apache. Even more significantly, the open source movement has largely permeated corporate software development. Google’s Android mobile phone platform, Sun’s OpenOffice office suite or, to a lesser extent, Apple’s App Store bear witness to the phenomenal power of the collaborative effort of thousands of individuals.

The success of open source software is puzzling in many respects. Open source licences require that source code be disclosed and available to all for free. Some so called restrictive licences, such as the General Public License, even impose that any extension of open source software be in turn licensed under the same terms. These legal provisions confer open source software the characteristics of a public good, and make it all the more striking that a large number of developers contribute to develop it. Against this backdrop, research in economics has been mostly interested in highlighting the various reasons why firms and private developers do contribute (Lerner and Tirole, 2002; Bonaccorsi and Rossi, 2003a; Lakhani and Wolf, 2005).

In this chapter, we explore another key aspect of the success of open source, namely the process of software code production in open source projects. Using a panel of 10,553 open source projects tracked on the SourceForge repository website from February 2005 until June 2007, we estimate the individual productivity of voluntary and corporate developers, as well as spillovers flowing from other projects. Our choice of a flexible translog specification of the project production function makes it possible to assess scale effects and interactions between voluntary and corporate developers.

Our empirical analysis sheds light on several important features of open source production. We find that corporate developers are more productive than voluntary ones. However a negative interaction term indicates that associating the two groups entails a decrease in productivity. Our results also suggest that the way open source projects are organized makes it possible to manage large projects quite efficiently. Indeed the production of open source code is not subject to strong decreasing returns to scale, thereby contrasting with proprietary software production.

Finally, we provide evidence of knowledge spillovers flowing between projects of the SourceForge repository. The possibility to reuse code from other projects is frequently presented as a strong advantage of the open source model (Raymond, 2001). Our findings confirm this statement: the amount of software code available in other projects substantially increases a project's productivity. Spillovers benefit mainly to mature projects, and take place between projects that share the same topic and/or programming language.

Moreover, it appears that corporate programmers are better able to exploit spillovers than voluntary ones.

This work is one of the first empirical studies on the open source production process. One other paper estimates a production function based on SourceForge projects (Giuri et al., 2010).

Three other papers, all of which using data from the SourceForge repository, investigate related topics like sorting of programmers and R&D spillovers. Belenzon and Schankerman (2008) use the license types to infer the motives of programmers. They find in particular that corporate sponsorship – reported through a separate survey – tends to attract more contributions by developers under a less-restrictive license, and does not affect contributions by developers under a restrictive license. Using a different method (based on the email addresses of developers) to track corporate involvement, we complement their work by finding evidence of a productivity loss when corporate developers interact with voluntary ones.

Taking the number of downloads as a measure of outputs, Fershtman and Gandal (2008) show in turn that substantial spillovers flow within networks of projects and networks of developers. In a similar vein, Singh et al. (2008) focus on spillovers due to personal relationships among developers. Relating source code contributions with repeated interactions, they argue that trust and cohesion determine the size of the knowledge spillovers and thus affect the success of a project. By contrast, we propose another approach of spillovers, which is directly based on the availability of source code. Using source code releases as a measure of output, we take the quantity of code available in other projects as the source of spillover flows. We can thereby relate these flows with the projects' development stage, topic, programming language, and category of programmers.

The chapter is structured in seven sections. We review in Section 2 the economic literature on open source production. Next, we develop our regression model (Section 3) and, in turn, present our data (Section 4). We discuss our results in Sections 5 and 6, the latter being specifically devoted to spillovers. Section 7 concludes.

2.2 Background

This section reviews the literature on open source software production. After quickly presenting the incentives of developers, we discuss the organization of code production within projects and the effects of knowledge spillovers in open source software.

2.2.1 Programmers

Traditionally, open source developers have been volunteers who contribute code for individual motives. Studies find that between 59% (Lakhani and Wolf, 2005) and 49% (Ghosh et al., 2002) of surveyed open source contributors are unpaid volunteers. They submit pieces of source code for a variety of reasons. Lakhani and Wolf (2005) find that 59% of their respondents contribute because they need a specific feature. Some also contribute because it is fun to solve challenging programming tasks (Shah, 2006), for ideological motives opposing free software to proprietary software, or to improve or signal their programming skills (Lerner and Tirole, 2002).

Apart from private volunteers, firms increasingly participate in open source development as well. According to a recent survey, they develop 15% of all open source software world-wide (UNU-MERIT, 2006). More strikingly, Lakhani and Wolf (2005) find that around 40% of surveyed open source developers are paid to contribute to projects. There are different reasons why firms *use* open source software. It may be a cheaper alternative to in-house software development or buying proprietary software (Eilhard, 2008b). Firms can also make profits by combining it with the proprietary code, hardware or services they supply to consumers (Lerner and Tirole, 2005; Henkel, 2006). Why firms *contribute* code is a more difficult question. Available surveys highlight various motives. Besides complying with open source licenses, these motives especially include being accepted by the community of developers and influencing the development path of the software (Henkel, 2006).

It is unclear whether corporate programmers are more productive than voluntary ones.

Empirical evidence shows that voluntary and corporate contributors spend approximately the same amount of time on working in open source projects (Ghosh et al., 2002), although we do not know exactly whether they allocate their time equally on the same types of projects. Differences in individual productivity may also depend on skills. Corporate programmers are paid professionals, while many voluntary programmers contribute to projects to demonstrate or improve their skills (UNU-MERIT, 2006). Lamastra (2009) investigates the impact of corporate developers on the quality of SourceForge projects. She finds that corporate involvement increases development activities, but reduce the quality of all tasks not related to development.

2.2.2 Organization

The most striking feature of open source production being that it is realized by communities of peers (von Hippel and von Krogh, 2003), several authors seek to understand how coordination works within each project. Lerner and Tirole (2002) emphasize the existence of informal hierarchies in open source projects. Each project is organized in circles of more or less involved developers (from those who report bugs to those who fix them) around one or several 'charismatic' leaders recognized as such by their peers. By contrast, Benkler (2002) contends that the organization of open source production differs from both the hierarchical and market models. In his view, labour division between small contributors hinges on the combination of voluntary contribution and modular software design, which the development of information and communication technologies has made possible.

Arguably, this productive model can ensure a more efficient division of labor than traditional ways of developing software code (Bonaccorsi and Rossi, 2003c). The modular design makes it possible for volunteers to select for themselves on which problems they want to work, without having to care about the other modules. Matching their skills with the tasks at hand, they moreover take up tasks which more closely fit their set of capabilities (Amabile, 1983). Self-assignment thus can lead to a better use of human capital in open source projects and hence to a higher effectiveness compared to traditional software production (von Krogh et al., 2003).

In a recent empirical study based on SourceForge projects, Belenzon and Schankerman (2008) find that the sorting of programmers also strongly depends on the license type, project's size and corporate sponsorship. Interestingly, they show that the effect of the license type on contributions varies with the project type: a restrictive license will draw more contributions for end user projects, and less for developer tool projects.

2.2.3 Firms' involvement in open source projects

Cooperation between voluntary and corporate developers is of particular interest. The open innovation paradigm (Chesbrough, 2003) suggests that complementarities in skills and objectives may further boost open source production. Firms are for instance more interested in improving user interfaces or interoperability with hardware – two dimensions which are usually neglected by voluntary user-developers (Lerner and Tirole, 2002; Scotchmer and Maurer, 2006). Yet corporate developers may also be reluctant to share source code if the competitive advantage of their firm is at stake (Henkel, 2008). As a result, they practice various forms of selective revealing to protect valuable information from leaking to open source projects (Bonaccorsi and Rossi, 2003b; Dahlander and Wallin, 2006; **?**).

Reflecting the conflict of interest, Henkel (2006) observes that 51% of surveyed firms do not reveal their contributions to open source projects at all. Still, many firms contribute to open source projects, some of them intensively (Dahlander and Magnusson, 2005). Beyond compliance with open source licenses, they do so to be able to influence the project's development path, which in turn requires being perceived as a good player by voluntary programmers (Henkel, 2006). Accordingly, corporate developers tend to work in key positions in open source projects, and to interact more frequently than others with core developers (Dahlander and Wallin, 2006).

2.2.4 Spillovers

Spillovers are another aspect of the open innovation paradigm. As stated by Raymond (2001): “Good programmers know what to write. Great ones know what to rewrite (and reuse).” Indeed source code under open source licenses being publicly available, the stock of open source software has become a wide library in which programmers are free to pick and reuse the pieces of code they need for a particular project. Put in economic terms, the knowledge – the code – generated for each projects spills over to other projects.

Although spillovers may be a key factor in the spectacular growth of the open source movement, empirical studies have been missing until very recently. Fershtman and Gandal (2008) focus on spillovers flowing through projects involving the same developers, and through developers involved in the same project. They show that a project indeed benefits of spillovers when (i) it is directly connected to other projects (ii) it is close to a large number of projects beyond their immediate connections (iii) it occupies a central position in a network of projects. Taking a closer look at contributors, Singh et al. (2008) explore spillovers which accrue through social interaction. They argue that projects are more successful when developers have (i) a high degree of internal cohesion, which leads to mutual trust and reciprocity norms, and (ii) a moderate level of relationships with developers outside the project. This suggests that, while information flows among developers are important, spillovers are not restricted to projects that share common developers.

2.3 Econometric Model

The measurement of open source projects' productivity encounters the same type of difficulties as any attempt to measure the true productivity of research (Hausman et al., 1984). Ideally, we would need a measure of the social return of each project, including the utility users derive from the software as well as the spillovers to other projects. Unlike corporate research, the prices of innovative goods cannot proxy spillovers, let alone users' utility, for open source software is distributed royalty free. It might be more realistic to infer private returns, namely the returns for contributors, from their degree of individual involvement. Such a work would require precise information on their individual productivity and the time they spend on each project.

Our objective in this paper is more modest and constitutes a first step in that direction. We focus on the determinants of technical success in open source software development, as measured by software releases. Such releases signal that a new version of the open source software is ready for download. They provide an interesting proxy of innovative output since released versions are generally improvements upon the previous versions which they replace.

Releases have another advantage over other methods to measure software production, e.g. lines of code or function points (Boehm, 1981; Wheeler, 2002). They have a straightforward economic interpretation. While more technical measures track software creation more precisely, they do not readily translate into new software products. Apart from the problem of comparing projects with different programming languages, there is a more important issue here: Software developers regularly add and remove lines of code from the source code. How can we measure developer productivity with this? Are developers more productive simply because they write more lines of code? If we take the analogy of a creative writer, we see that these technical measures miss the point for an economic analysis. Creative writers are not economically productive because they write long novels. Writers are productive when they publish books. The same holds for software developers. We, as economists, are not interested in the elegance of the source code, in contrast to computer scientists. We want to measure the working pieces of software that the open source community produces during a specific time period.

We have no priors about the true functional form of the open source production function. Since our dependent variable, the number of releases for project i in each time period $t = 1, 2, ..., 28$, Y_{it}, only takes on non-negative integer values, we assume that it is generated by a Poisson process. We model the single parameter of the Poisson distribution function, λ_{it}, as a function $F(X_{it}, \beta)$ where X_{it} is a vector of labor inputs for each project

i in time period t and its respective parameters, β. We also add a set of control variables, P_{it}, which includes changes in project topic, in development stages, licenses and time dummies per month. We use a project-level fixed effect estimation. The term α_i captures all time-constant, unobservable effects, ϕ_t contains time dummies for each month and ϵ_{it} includes unobservable, time-variant productivity factors.

$$ln\,(Y_{it}) = ln\,(\lambda_{it}) = F(\cdot) + P_{it}\,\delta + \alpha_i + \phi_t + \epsilon_{it} \tag{2.1}$$

To reduce clutter, we drop the subscripts for projects and time periods in the further presentation when the interpretation is unambiguous. We want to analyze the productivity of the three different groups of contributors. Without any prior constraints on the function $F(X, \beta)$, we estimate the coefficients of a translog production function for all open source projects. The translog function is an attractive flexible specification. It has both linear and quadratic terms with the ability of using more than two factor inputs, and can be approximated by a second order Taylor series (Christensen and Greene, 1976). Applied to our three labor inputs, the translog production function can be written in logarithmic form as:

$$F(N_{k,l}, S) = \sum_k \beta_k\, ln\, N_k + \frac{1}{2}\sum_k \sum_l \beta_{kl}\, ln\, N_k\, ln\, N_l + \gamma\, S_{it} \tag{2.2}$$

where $N_{k,l}$ is the respective number of developers for academic, corporate or private developer groups, $k, l = \{A, C, P\}$, and S_{it} is a spillover proxy which we will define later in the chapter.

Observe that the translog function can be transformed in a standard Cobb-Douglas function by imposing zero-value coefficients on the second order terms. This would imply that the estimated elasticity of output with respect to each input be constant by assumption. For sake of comparison, we also estimate this constrained specification.

Besides its flexibility, another advantage of the translog function lies in the interpretation of the second order terms. The sign of the coefficients for the squared logarithms of labor inputs hints towards increasing or decreasing elasticity of output. We can therefore use it as an indicator of increasing or decreasing returns with respect to each category of programmers. Moreover, positive or negative coefficients for the interaction terms, $ln\, N_k\, ln\, N_m$, make it possible to derive conclusions on the effect of interactions between different categories of developers.

Estimating a production function creates a simultaneity problem (Marschak and Andrews, 1944; Olley and Pakes, 1996). In the traditional setting, firms choose the factors of production according to profit-maximizing, first-order conditions. Because they consider firm-specific productivity differences, more productive firms use more factors of production, rendering it more difficult to establish a causal link between production factors and output. The control variables in P_i are a first way to capture differences in productivity between projects. These variables include dummies for changes in the development stage of the projects and their topic of application, and several channels of knowledge spillovers from other projects. We moreover address the simultaneity problem by using a project-level fixed-effect estimation method. This approach eliminates major misspecification, which are transmitted to the factor decisions and are constant over time (Griliches and Mairesse, 1998). It however requires the assumption that unanticipated elements of the error term at period t do not affect factor decisions at later periods.

2.4 Data

2.4.1 Data Source

We use a balanced panel of 10,553 open source projects tracked on SourceForge from February 2005 until June 2007 (T = 28 months). SourceForge is an internet platform for open source projects. It helps new projects attract developers and users. Sourceforge provides the necessary tools for managing a software project, such as user fora, bandwidth for downloads and a version control system to keep track of contributions. SourceForge is the largest internet repository with almost 300,000 projects, similar, but considerably smaller, platforms are GoogleCode or Kenai.

SourceForge as an incubator includes a large number of small or inactive projects. Since creating new projects is costless, many projects may be started, just to be subsequently abandoned after a short while. For this reason Howison and Crowston (2004) caution the use of SourceForge projects to draw conclusions about open source projects in general. In this regard our approach is especially valuable, because we only look at projects which have already released a version of their software, thus avoiding inactive projects all together.

Open source projects regularly release newer versions of their software. These file releases fix programming bugs, add new features or improve performance. Larger projects generally announce their releases in advance; smaller projects may post updates intermittently. There are various forms in which users can obtain these releases: compressed source code, binary installers or text files. To have comparable observations, we only look at compressed files. We only look at gzip files, as these were the most common types of compressed files (30%) in the sample. This allows us to look at projects across different operating systems, because binary installers often are operating system specific.

In addition to information on software projects, we obtained developer backgrounds using their email addresses. These email addresses were related to the corresponding websites and sorted into three categories: academic, private and corporate. We sampled ten websites to obtain keywords for each category and counted the occurrences of each word. We established a ranking and were thus able to attribute an email address with either category, for example 'hotmail.com' to the private category, 'ibm.com' to the corporate and 'ensmp.fr' to the academic one. In all, we checked approximately 15,000 websites. We then retained projects where all developer backgrounds were known as well those which actually released a file update during the observed time period. so we could account for all registered developers in a project, 10,553 projects remained in our sample. Lerner et al. (2006) use a similar method in their study. Their analysis encompasses the top-level domain names, i.e. '.com' for corporate developers or '.edu' for academic ones. This has clear disadvantages compared to our method, because top-level domain names, especially for '.com', confound private email accounts, such as for 'yahoo.com', with corporate ones.

Our method has two drawbacks. On the one hand, we might see a type I error. We overestimate the number of corporate developers for those who use their work email for private purposes. They subscribe to SourceForge using their office email and thus pass as corporate developers, even though their company does not directly promote open source development. On the other hand, we might see a type II error. Contributors might use their private email addresses for subscription, although their firms do support open source projects. There is no direct way to mitigate these issues. One possibility is to argue by deduction. Finding statistically significant differences between corporate, private and academic developers might support the validity of our data collection method.

2.4.2 Descriptives

SourceForge offers 222 different topics for open source software. After regrouping these categories in broader application fields, we obtain a more manageable 19 topics. Figure 2.1 shows the number of projects per topic in our data sample. We immediately see that a lot of projects focus on four topics: software development, internet, communications and system administration. The figure also shows the tremendous variety in topics that we find on SourceForge. There are projects on religion - for instance an application to calculate the Islamic prayer times - as well as scientific software and games - incidentally, approximately 200 versions of Tetris are available on Sourceforge.

The number of developers is a possible indicator for project size. Figure 2.2 shows that our sample contains an overwhelmingly large number of small projects. We see that almost 80% of the observed projects have only one registered developer over the entire time period. This begs the question whether these project have a significantly different mode of production than projects with more developers. To test the impact of this possible

Figure 2.1: Projects per Topic

bias, we will estimate the production function with a sub-sample of the projects with at least two developers (n' = 2,406). A more detailed look at the developer categories reveals that on average each project has at least one private developer and every fifth project has a corporate one. Also, every fifth project has an academic contributor (Table 2.1).

Projects release new updates irregularly. Figure 2.3 depicts the number of projects for the maximal number of file releases within the 28 months. We see that the majority of projects has only one release over the entire period and that there are few projects which are very productive, having more than ten releases.

We measure spillovers with the size of the code commons, i.e. size of the available knowledge base. We compute the code commons available to project i as the sum of the number of bytes of the new releases at time t, $S_{it} = \sum_{j \in J_m}(Bytes_{jt})$, for all other projects J, where $i \notin J$, in a particular topic m, $J_m = \{J_1, J_2, ..., J_{19}\}$. We consider topic to be a meaningful boundary of the code commons, because it encompasses software with similar objectives, similar problems and, likely, similar solutions. Following the same reasoning, we also calculate the available code commons with respect to topics and programming languages. We assume that Table 2.1 shows that average size of the code commons is considerable. Projects have on average 81 terabytes of - compressed - source code available to copy and reuse. The refined knowledge base of projects with same topic

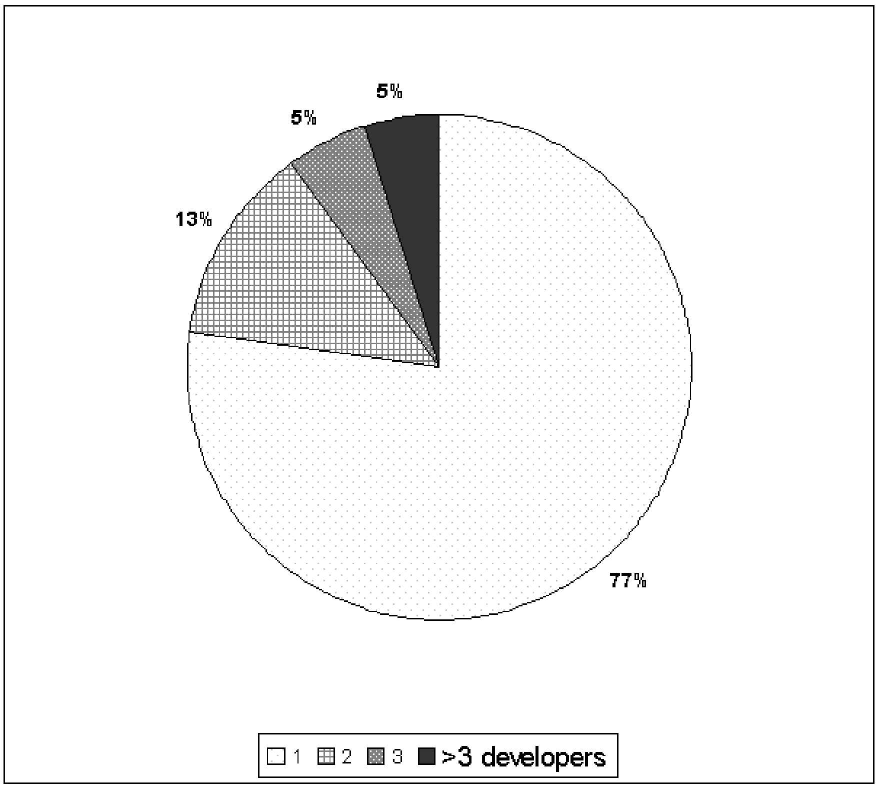


Figure 2.2: Maximal Number of Developers per Projects

Table 2.1: Summary Statistics

Variable	Mean	Median	SD	Min	Max	Observations (n*T)
Academic Developers	0.200		0.650	0	12	295,484
Corporate Developers	0.223		0.587	0	12	295,484
Private Developers	1		0.827	0	15	295,484
File Releases	1.347		1.198	0	21	295,484
Spillovers - same Topic (Tbytes)	80.5	22.0	163.3	0	611.1	295,484
Spillovers - same Topic/same Programming Language (Tbytes)	9.9	1.1	35.7	0	410.0	295,484
Spillovers - same Topic/diff. Programming Language (Tbytes)	70.6	17.1	148.5	0	611.1	295,484
Spillovers - different Topic/same Programming Language (Tbytes)	85.03	51.1	118.84	0	515.76	295,484

and same programming language still encompasses almost 10 terabytes of source code which is about the size of the printed collection of the U.S. Library of Congress.

SourceForge provides a way to measure the development status of each project. Project administrators indicate in which of the seven different stages the project is: planning, pre-alpha, alpha, beta, stable, mature or inactive. Although the choice of one stage over another is based on subjective criteria, it gives us a general idea of the progress and stability of the project. Table 2.2 shows the frequencies of projects within each development stage. The mean gives the percentage share of each category within our sample. We see

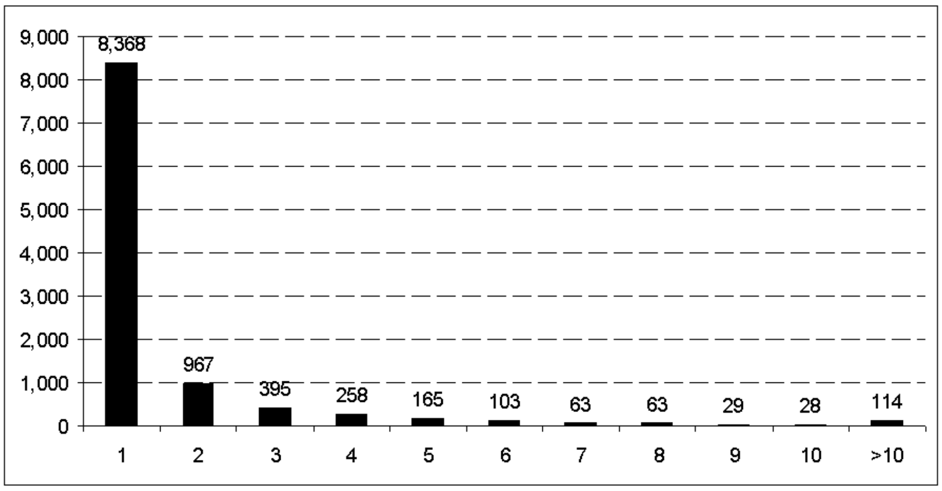


Figure 2.3: Number of Releases

Table 2.2: Development Stages

Variable	Mean	SD	Min	Max	Observations (n*T)
Planninng	0.04	0.18	0	1	295,484
Pre-Alpha	0.12	0.32	0	1	295,484
Alpha	0.20	0.40	0	1	295,484
Beta	0.31	0.46	0	1	295,484
Production	0.28	0.45	0	1	295,484
Mature	0.02	0.14	0	1	295,484
Inactive	0.00	0.06	0	1	295,484

that around 60% of the projects in our sample are in beta or later stages. This provides further evidence that we are indeed dealing with actively developed projects.

2.5 Discussion

Table 2.5 shows the results of the regressions. Models 1 and 2 show the estimates for the full sample (n = 10,553). We progressively render the functional form more flexible, beginning with a classic Cobb-Douglas production function (model 1) and using the complete translog specification (model 2). The same translog specification is estimated in models 9 and 10, respectively for the sample of projects with at least two developers (n' = 2,406) and for projects in beta or later development stages (n" = 7,018). All regressions are fixed-effects models. A Hausman test shows that random-effects estimates are inconsistent, refuting the assumption that unobservable project characteristics are

Table 2.3: Likelihood Ratio Test

	Chi^2-Value	p-Value
Model 1 (Cobb-Douglas)	224.31	0.000

uncorrelated with past, present or future values of the regressors. Considering the nature of the independent variable, the count of file releases, the use of a Poisson panel regression lends itself rather directly. The absence of zeros in the dependent variable prevents the use of negative binomial regressions to test for overdispersion. We argue instead by visual inspection (see table 2.1) that the assumptions for a Poisson regression are met, namely that $Mean(Releases) = Var(Releases)$.

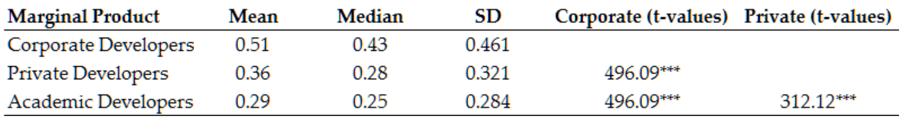
We check the validity of our specification with a likelihood ratio test. Comparing the complete translog function (model 2) with a Cobb-Douglas specification (model 1), we test the null hypothesis that the each restricted model is a more adequate representation of the data than the translog function. Table 2.3 shows the chi-squared and the respective p-values in parentheses. The null hypothesis cannot be supported, thus suggesting that indeed model 2 is the best specification for our data.

2.5.1 Developer Productivities

As can be seen in Table 2.5, we find significant coefficients for each category of developer in the Cobb-Douglas specification (model 1). Recall that by construction this specification implies constant elasticity of output with respect to each input. The estimated input coefficients measure these elasticities. They suggest a slightly lower elasticity for corporate programmers than for private and academic ones. However, this conclusion is not confirmed statistically, the coefficients being equal with a high probability (p-value $= 0.85$).

Model 2 makes it possible to go further in the comparison. The translog function relaxes the assumption of constant elasticity by taking into account additional input variables, namely the squared logs and the crossed logs of developers. This flexible specification also makes it possible to disentangle the different effects captured in each Cobb

Table 2.4: Computed Marginal Products



Marginal Product	Mean	Median	SD	Corporate (t-values)	Private (t-values)
Corporate Developers	0.51	0.43	0.461		
Private Developers	0.36	0.28	0.321	496.09***	
Academic Developers	0.29	0.25	0.284	496.09***	312.12***

Douglas coefficient. Although the simple logs of developers remain significant in model 2, they have lower values than in model 1, and the estimated coefficient become now higher for corporate developers. Moreover, significant and positive coefficients for squared logs show that elasticity of output with respect to each category of developer is in fact increasing. In line with simple logs, these coefficients are higher for corporate developers than for the other two categories. As shown in Table 2.5, the crossed effects are significant only when corporate developers are involved, and then negative in each case. In other term, we find empirical evidence of a negative effect of interactions between corporate developers and other categories on the productivity of open source projects.

In sum, the productivity of corporate developers is subject to conflicting effects. On the one hand, coefficients of simple logs and squared logs suggest that corporate developers are more productive than other categories. On the other hand, corporate developers also have a negative effect on production when they are associated with other categories.

Making comparisons between categories of developers requires taking into account these conflicting effects. As a first step in this direction, we calculate the marginal product of each developer group for each project in model 2. Table 2.4 presents the average and the median computed marginal products for each developer group as well as the respective t-statistics for equality of means. We see that the average corporate marginal product is significantly higher than the marginal product for the other two groups. The difference is statistically significant at the 1% level.

Our results suggest that corporate developers are on average more productive than private or academic contributors. Table 2.4 shows that an additional corporate developer is on average 42% more productive than a private contributor and approximately 78% more productive than an additional academic developer. The median marginal productivity might mitigate outliers and give a more accurate picture of the productivities in our data sample. Considering the medians, the difference in productivities between contributor groups is even more pronounced. Here, adding a corporate contributor increases productivity by 54% more compared to an additional private developer and 72% more compared to an additional academic one.

2.5.2 Scale effects

Looking at the average marginal product gives only a partial picture, for this leaves out diminishing or rising marginal productivity. To address this limitation, we consider now the relationship between productivity and the scale of open source projects.

The simple Cobb-Douglas specification in model 1 makes it possible to derive clear conclusions on returns to scale. The sum of the estimated coefficients for each category of programmers is equal to 1.9813 – much above 1 – which suggests that open source production is subject to increasing returns to scale. The quadratic and interaction terms make it more difficult to identify returns to scale in the translog model. To address this problem, we simulate five scenarios based on model 2: Projects with (i) all corporate, (ii) all private, (iii) all academic, (iv) cooperation in which the number of contributors is equally divided between the developer groups and (v) cooperation where the number of developers is weighted among academic, private and corporate contributors with respect to their sample frequencies. Figure 2.4 presents these simulations. In the first three scenarios the project's output is clearly a convex function of the number of developers within the range of developers we observe. In other terms, each new contributor increases the average developer productivity.

Returns to scale are increasing for projects that have developers with the same background. The question though is how well do the 'all-with-one-background' scenarios correspond to our sample. These three scenarios are clearly polar cases and we need to look at what goes on in between these three extreme examples. We therefore calculate two more simulations in which we assume that developers are added from each developer group.

In Figure 2.4b, we can see that the curves for the fourth and fifth scenario are slightly concave. We add a 45° line to make the concavity easier to see. For scenario 4 (equal distribution), each new contributor is a less productive than the one before him. Average productivity decreases as more developers are added to a project. The decreasing returns to scale are less obvious in the fifth scenario (weighted distribution). Nonetheless, as more developers are added to a project, average productivity decreases slightly, as can be seen by comparing it with the 45° line.

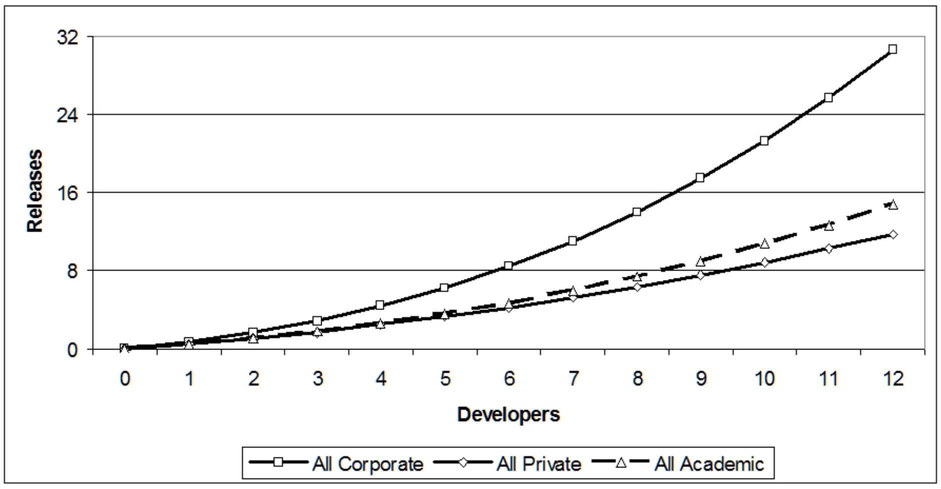
The last two scenarios show that open source projects exhibit slight, decreasing returns to scale. Our results depend on the functional form of the production function and on the assumed entry to open source projects of developers with different backgrounds. The heterogeneity of development communities determines whether we are more likely to observe open source development resembling scenarios 1-3 or scenarios 4 or 5. To be sure, all five scenarios are extreme cases that do not completely represent the actual productivity in open source development. These scenarios give us an indication on the average dynamics inside open source development projects.

Figure 2.5 represents the notion that these scenarios are the upper and lower bounds in which our model predicts the returns to scale. The upper bound is the “one-single-background” scenario for corporate developers and the lower bound is scenario 4 in which we add contributors from all three backgrounds equally. Scenario 4 is the lower bound, because the negative interaction effects between groups are largest in this scenario. The area in between these bounds shows the potential returns to scale for any distribution of contributor groups in a development community. Looking at the area between the two bounds, we can infer that there is a considerable potential for increasing returns to scale in open source projects.

A possible factor driving these results could have been a qualitative difference in nature between the releases observed in nascent and more mature project. Developing the first version of a software product may indeed take more time than subsequent improvement.

Figure 2.4: Simulations 1

(a) Same Background



(b) Cooperation

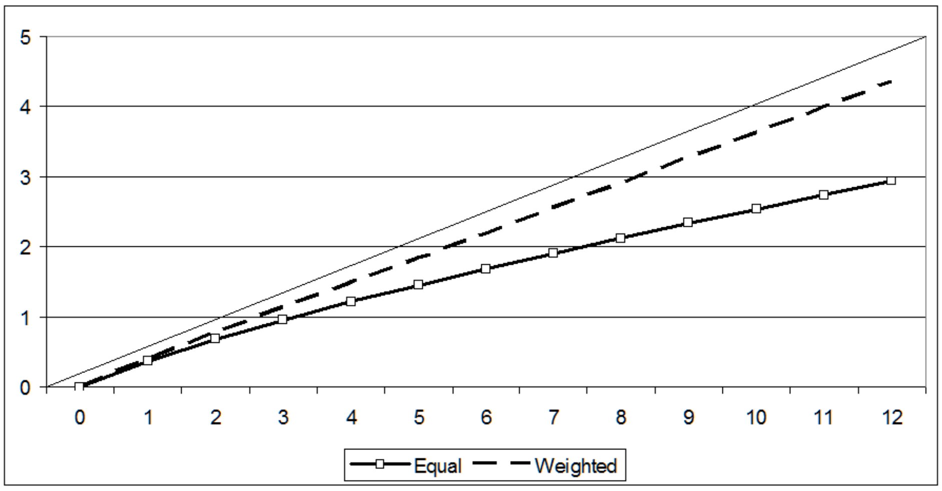
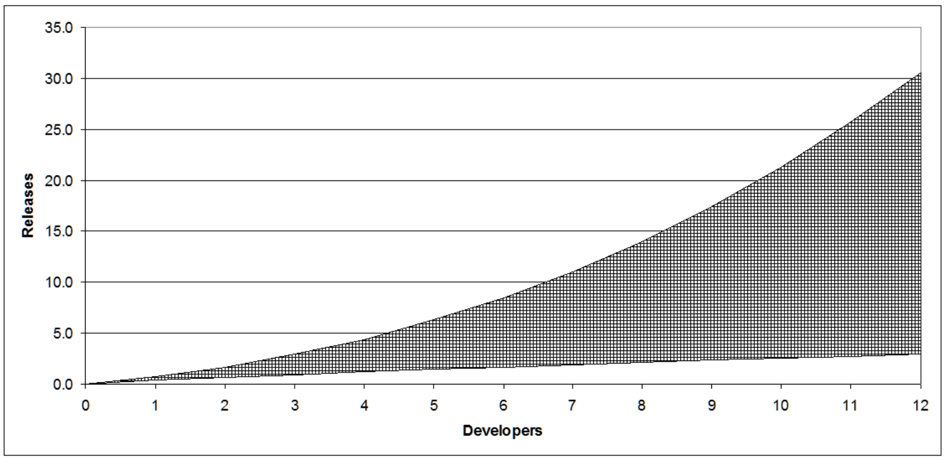


Figure 2.5: Simulations 2



We have therefore estimated the same translog specification respectively for the sample of projects with at least two developers (n' = 2,406) and for projects in beta or later development stages (n" = 7,018). As can be seen in Table 2.5 (models 9 and 10), the results do not differ substantially from model 2. Another possible explanation lies in the way we measure the developers' contribution. The increasing returns to scale we observe might reflect the fact that developers tend to devote more time on large mature projects – which we cannot control for. Still, our results indicate that open source production is hardly subject to decreasing returns. Despite a possible bias in time measurement, this sheds light on the intrinsic efficiency of this software production model. The combination of modular design and voluntary contributions seem to be an efficient way to divide labor, even for large projects.

2.5.3 Spillovers

In Tables 2.7 and 2.8, we present several models assessing the spillovers effects flowing between projects through various channels. They show that the coefficients for spillovers are positive and significant. The size of the code commons has a differentiated effect on production depending on the developer group, topic, programming language and the development stage of the project.

Model 2 includes a unique spillover variable denoting the number of terabytes of source code available in other projects of the same topic. Albeit small, we find a positive and significant coefficient for spillovers. Calculating the impact of the average input factors and spillovers on production, we find that spillovers account for 2.5%, whereas developers make out 97.5% of the entire effect (table 2.6).

We need to address the size of the spillovers on overall productivity. Why is the spillover effect so small? On the one hand, this may be due to the difference between codified information and tacit knowledge. In contrast to codified information, tacit knowledge can only be learnt through face-to-face interaction or experience. Tacit knowledge spillovers may be therefore less commensurable with our definition of code commons and are left out of our estimation. This measurement error can lead us to underestimate the effect of knowledge spillovers. On the other hand, it is also possible that the effect of knowledge spillovers on productivity is small compared to other factor inputs. As we said before, we conflate the influence of labor and capital inputs in our estimation. Compared to these, the overall gains from knowledge spillovers might be minute. In the end, software development may come down to having a powerful computer and spending a lot of manpower on a project.

Despite the small size of the spillover effect, it is worthwhile to look more closely at the channels through which spillovers flow. Looking at the interaction terms of developer groups and spillovers (model 3), we find that the coefficients are significant for corporate and academic contributors. The results indicate that adding another terabyte of code commons increases the productivities for these two developer groups, but not for the private one.

It thus seems that academic and corporate contributors copy and reuse more than private developers and hence benefit more from the available code commons. There is no significant difference between the effects for academic and corporate contributors ($\text{Chi}^2 = 1.02$). According to the adage that "good programmers know what to write. Great ones know what to rewrite (and to reuse)" (Raymond, 2001), it would appear that academic and corporate contributors are better open source developers than private ones. Of course, there is another way to read these results. One could argue that corporate and academic contributors are more pragmatic and goal-oriented than private ones. They may not write software for fun or to learn. Using available source code in other projects helps them achieve their objectives more quickly. This pragmatism might cause the larger spillover effect for academic and corporate developers.

Furthermore, Model 4 looks at the impact of the license type on spillovers. We find that the coefficient for same license type is positive, but not significant. By contrast the coefficient for different license type is positive and slightly significant. Sharing the same license type does not appear to promote knowledge spillovers, however the size of the knowledge base matters for projects from different license family. Leaving the differences in statistical significance aside - indeed at 10%, the latter coefficient is only slightly significant -, this finding might say that, once source code is made public, contributors care little about the license type and whether the license is restrictive or not. They may simply use the available source code regardless of the constraints on license proliferation. Hazarding a guess, we might say that there is a commingling of different license types at the source code level.

Model 5 and 8 show that the further advanced a project is in its development, the more it benefits from the available source code in other projects. Spillovers are significant only in projects with Production and Mature status. A Wald test shows there is a significant difference in spillovers between projects in Beta and Production stages (Chi^2 = 38.72). It thus appears that spillovers are not a critical resource to start new projects, but rather to improve and extend more mature projects. This suggests that spillovers could relate to bug fixing or generic functionalities rather than the core of new software. Since they mainly benefit the developers of mature projects, this result also provides an interesting explanation for the non-decreasing returns to scale we observe in SourceForge projects.

We fail to find a significant difference between spillovers from projects with the same topic and programming language and spillovers from projects with different programming languages in model 4. Separating the proxies further, however, shows that spillovers for projects with the same programming language are significantly stronger than for projects with different languages (model 6). A Wald test reveals that there is a significant difference between spillovers of same versus different programming languages in production-stage projects (Chi^2 = 4.45) and mature ones (Chi^2 = 7.6). Moreover, we find significant spillover effects from projects with the same programming language, but different topics in models 5 and 6. In model 5, the difference between the spillover effects is significant for same topic-different programming language spillovers and different topic-same programming language.

Finally, we find significant spillover effects from projects with the same programming language, but different topics in models 5 and 6. In model 5, the difference between the spillover effects is significant for same topic-different programming language spillovers and different topic-same programming language.

2.6 Conclusion

The purpose of this paper was to study empirically the production of open source software. Using a panel of 10,553 projects registered on SourceForge over a period of 28 months (February 2005 until May 2007), we have estimated a production function relating the number of file releases with the number of corporate, private and academic contributors. We have considered two possible specifications of the production function, namely a Cobb-Douglas function and a more flexible Translog specification. This approach made it possible to highlight various interesting results, concerning the productivity of corporate and other developers, the effects of their interactions within open source projects, returns to scale driven by labor division and the existence of spillovers between projects.

Our first findings concern the developers' productivity. We find empirical evidence that corporate developers are generally more productive than voluntary ones. However, this result must be balanced with the negative effect of interactions between the two categories of developers. In other terms, although corporate developers are more efficient individually, they seem less efficient in cooperating with other categories of developers within a given project, which suggest possible conflicts or at least coordination failures due to vested interests and/or different ways of approaching their work.

Our estimations suggest that open source projects may not be subject to decreasing returns to scale. This result should be taken with caution due to a lack of data on the time developers spend on each project. Still, it denotes a striking constrast with decreasing returns that are oberserved in more traditional software production (Brooks, 1978). This suggests that the peculiar organization of production in open source projects – based on a combination of user-driven innovation, voluntary contribution and modular design –

enables an efficient division of labor between programmers. Such organizational efficiency may explain why large open source projects such as Linux or Apache are particularly fierce competitors for proprietary software.

Finally, we find evidence for positive spillovers between projects of the Source Forge repository. We test different possible channels, and show that spillovers mainly flow between projects with the same topic and, to a lesser extent, between projects with the same programming language. Projects involving corporate and academic developers are more likely to benefit from spillovers. Project development stage also matters: spillovers only seem to benefit mature projects, thereby increasing further the total productivity of large projects. Surprisingly, the license type neither favors nor hinders the flow of spillovers between projects.

Table 2.5: Main Results

Releases	**Model 2**	**Model 1**	**Model 10**	**Model 9**
Corporate Developers (ln)	0.5477***	0.6259***	0.5240***	0.5252***
Private Developers (ln)	0.4033***	0.6675***	0.4296***	0.3726***
Academic Developers (ln)	0.3277*	0.6679***	0.1547	0.278
Corporate Developers (ln)^2	0.3114***		0.2991***	0.2738***
Private Developers (ln)^2	0.2292***		0.1951***	0.1999***
Academic Developers (ln)^2	0.2918***		0.3749***	0.2760***
Corporate/Private Developers	-0.2569**		-0.2637**	-0.2390**
Corporate/Academic Developers	-0.1994***		-0.2324***	-0.1849***
Private/Academic Developers	-0.0844		0.0139	-0.0764
Spillovers - same Topic	0.00014***	0.00014***	0.00014***	0.0001
Topic: Database	-0.0772**	-0.0763**	-0.1354***	-0.0257
Topic: Desktop	-0.0155	-0.0154	-0.0116	0.0176
Topic: Education	0.0613	0.061	0.1134**	-0.0781
Topic: Format	-0.0166	-0.0169	-0.0082	-0.0849
Topic: Games	0.0031	0.0033	-0.0154	-0.0024
Topic: Internet	-0.0550***	-0.0567***	-0.0393	-0.1211***
Topic: Multimedia	-0.0029	-0.0032	-0.0025	0.0032
Topic: Office	-0.0414	-0.0423	-0.0308	-0.0897*
Topic: Others	-0.0817	-0.0812	-0.0926	-0.1885*
Topic: Printer	0.0593	0.0598	0.0674	0.0093
Topic: Religion	-0.1457	-0.1457	-0.178	0.2153
Topic: Science	-0.1502***	-0.1483***	-0.1419***	-0.0732*
Topic: Security	0.0122	0.0119	0.0305	-0.1493***
Topic: Sociology	-0.0454	-0.0462	-0.123	-0.0898
Topic: Software Development	0.018	0.0164	0.0232	---
Topic: System	-0.0612**	-0.0588**	-0.0367	-0.0601
Topic: Terminal	0.1041	0.1047	0.1019	-0.0921
Topic: Text Editors	-0.0443	-0.0439	-0.045	-0.0596
Pre-Alpha Status	-0.0198	-0.0203	---	0.0103
Alpha Status	0.1301***	0.1284***	---	0.1089*
Beta Status	0.2779***	0.2773***	0.1156***	0.2337***
Production Status	0.5865***	0.5870***	0.4433***	0.4707***
Mature Status	0.7309***	0.7350***	0.5499***	0.4023***
Inactive Status	-20.074	-20.0747	-20.2538	-18.153
Time Dummies	sig.	sig.	sig.	sig.
Number of obs	295,484	295,484	191,017	67,368
Number of groups	10,553	10,553	7,018	2,406
Wald chi2(16)	9,191	9,149	8,329	3,879
Log likelihood	-297,089	-297,121	-197,029	-70,596

*: p < 10% **:p < 5% ***: p < 1%

Table 2.6: Impact of Average Input Factors

Total Effect	**Labor Factors**	**Spillovers**
100%	97.5%	2.5%

Table 2.7: Results for Spillovers 1

Releases	Model 5	Model 4	Model 3	Model 2
Corporate Developers (ln)	0.5588***	0.4481***	0.5793***	0.5477***
Private Developers (ln)	0.4060***	0.3009***	0.4000***	0.4033***
Academic Developers (ln)	0.3122*	0	0.3130*	0.3277*
Corporate Developers $(\ln)^2$	0.2883***	0.2214**	0.2487***	0.3114***
Private Developers $(\ln)^2$	0.2287***	0.2181***	0.2300***	0.2292***
Academic Developers $(\ln)^2$	0.2992***	0.2227**	0.2900***	0.2918***
Corporate/Private Developers	-0.2509**	-0.2	-0.2366**	-0.2569**
Corporate/Academic Developers	-0.1976***	-0.1198*	-0.2145***	-0.1994***
Private/Academic Developers	-0.0847	-0.0156	-0.0883	-0.0844
Spillovers - same Topic	-0.0001		0	0.00014***
Spillovers - same Topic / Corporate Developers			0.0003***	
Spillovers - same Topic / Private Developers			0	
Spillovers - same Topic / Academic Developers			0.0002**	
Spillovers - same Topic / same License		0.0002		
Spillovers - same Topic / different License		0.0002*		
Spillovers - same Topic/ Pre-Alpha status	-0.0002			
Spillovers - same Topic/ Alpha status	-0.0001			
Spillovers - same Topic/ Beta status	0.0002*			
Spillovers - same Topic/ Production status	0.0006***			
Spillovers - same Topic/ Mature status	0.0006***			
Spillovers - same Topic/ Inactive status	-0.0015			
Pre-Alpha Status	0.0136	0.0199	-0.0196	-0.0198
Alpha Status	0.1572***	0.0819***	0.1299***	0.1301***
Beta Status	0.2838***	0.1993***	0.2781***	0.2779***
Production Status	0.5595***	0.4644***	0.5872***	0.5865***
Mature Status	0.6798***	0.5914***	0.7334***	0.7309***
Inactive Status	-19.9702	-19.3951	-20.0743	-20.074
Topic Dummies	part. sig.	part. sig.	part. sig.	part. sig.
Time Dummies	sig.	sig.	sig.	sig.
Number of obs	295,484	273,583	295,484	295,484
Number of groups	10,553	10,531	10,553	10,553
Wald chi2(16)	9,343	6,060	9,205	9,191
Log likelihood	-297,008	-273,906	-297,079	-297,089

*: $p < 10\%$ **: $p < 5\%$ ***: $p < 1\%$

Table 2.8: Results for Spillovers 2

Releases	Model 8	Model 7	Model 6
Corporate Developers (ln)	0.5650***	0.5521***	0.5478***
Private Developers (ln)	0.4090***	0.4063***	0.4034***
Academic Developers (ln)	0.3116*	0.3281*	0.3276*
Corporate Developers (ln)^2	0.2830***	0.3090***	0.3114***
Private Developers (ln)^2	0.2271***	0.2281***	0.2292***
Academic Developers (ln)^2	0.3002***	0.2919***	0.2918***
Corporate/Private Developers	-0.2584**	-0.2619**	-0.2569**
Corporate/Academic Developers	-0.1965***	-0.1996***	-0.1994***
Private/Academic Developers	-0.0854	-0.0849	-0.0844
Spillover - same Topic/same Programming Language	-0.00076*	0.00015**	0.00014**
Spillover - same Topic/ Programming Language / Pre-Alpha stage	0.00004		
Spillover - same Topic/ Programming Language / Alpha stage	0.0004724		
Spillover - same Topic/ Programming Language / Beta stage	0.0006354		
Spillover - same Topic/ Programming Language / Production stage	0.00138***		
Spillover - same Topic/ Programming Language / Mature stage	0.00187***		
Spillover - same Topic/ Programming Language / Inactive stage	0.0021		
Spillover - same Topic/different Programming Language	-0.000041	0.00015***	0.00014***
Spillover - same Topic/diff. Programming Language/ Pre-Alpha stage	-0.00018		
Spillover - same Topic/diff. Programming Language/ Alpha stage	-0.00015		
Spillover - same Topic/diff. Programming Language/ Beta stage	0.00017		
Spillover - same Topic/diff. Programming Language/ Production stage	0.00045***		
Spillover - same Topic/diff. Programming Language/ Mature stage	0.0004**		
Spillover - same Topic/diff. Programming Language/ Inactive stage	-0.0022		
Spillover - different Topic/same Programming Language	0.000049**	0.00004**	
Pre-Alpha Status	0.0115	-0.0204	-0.0198
Alpha Status	0.1557***	0.1293***	0.1301***
Beta Status	0.2822***	0.2762***	0.2779***
Production Status	0.5572***	0.5846***	0.5866***
Mature Status	0.6818***	0.7291***	0.7309***
Inactive Status	-19.9647	-20.0742	-20.0739
Topic Dummies	part. sig.	part. sig.	part. sig.
Time Dummies	sig.	sig.	sig.
Number of obs	295,484	295,484	295,484
Number of groups	10,553	10,553	10,553
Wald chi2(16)	9,343	9,191	9,311
Log likelihood	-297,008	-297,089	-297,023

*: p < 10% **:p < 5% ***: p < 1%

Chapter 3

Loose Contracts, Tight Control: Corporate contributions in SourceForge projects

3.1 Introduction

Companies increasingly contribute in the development of open source software. A recent study shows that 70% of the development work for the Linux kernel has been done by corporate developers (Kroah-Hartman et al., 2009). To be sure, this is not a singular phenomenon. Firms are responsible for a considerable share of the development work in all kinds of open source projects. Lakhani and Wolf (2005) find that around 40% of open source developers are paid to contribute in projects found on the SourceForge website, while Ghosh et al. (2002) observe that 54% of their surveyed open source developers are paid.

But what leads firms to contribute in open source software? The academic literature proposes several answers to this question (Wichmann, 2002b; Henkel, 2006). Firms contribute in open source software to promote standardization. Contribution thus becomes a collaborative standardization process. Each firm contributes to have its specifications included in the final standard. In a similar vein, companies might send programmers to open source projects to ensure compatibility with their products. Research also shows that firms contribute to gain reputation within a development community. Lastly, firms contribute for strategic reasons, for example to undermine a dominant incumbent.

We propose in this chapter a complementary approach to the standards and compatibility reasoning. After all, the development community creates and improves software voluntarily. So why doesn't the firm simply request that a standard be included in a piece of software or an application be made compatible to a product? We argue that it contributes to mitigate the risks of the open source license. By using the open source software the firm signs an incomplete contract, the open source license. The lack of fixed feature specifications, delivery dates or development priorities render the software license incomplete and create contractual hazards for the company. Therefore, it imposes additional governance structure in its relation with open source communities by dedicating developers to the software project.

We use a two-fold empirical method to shed light on this issue. Our first empirical looks at 2,643 open source projects extracted from Sourceforge to establish a relation between contractual incompleteness and corporate participation. We find quantitative evidence of to the notion that firms face contractual incompleteness when dealing with open source communities. Our results suggest that projects are indeed more likely to have high corporate participation when they show high degrees of contractual hazards.

In particular, the longer it takes for projects to treat requests and the more variable the treatment is the higher is the share of corporate developers in the project. Supporting the incomplete contracts hypothesis further, we find that as the frequency of the treated submissions increases, corporate open source participation decreases. This is in line with the transaction cost literature which asserts that the rate of repeated transactions can mitigate the problem of incomplete contracts.

An exploratory survey provides additional insights on the attitudes towards firms within different development communities. From May until July 2009, we conducted a short email survey among 7,914 project maintainers who coordinate the development work in open source communities and obtained 60 valid responses. Their comments consistently mention the unwillingness of the community to pay special attention to corporate needs. This qualtitative evidence suggests that there can be merit in using the theory of transaction cost and incomplete contracts on corporate open source software. By participating, firms can affect the structure of the transaction with the development community and can more closely control their exposure to contractual incompleteness.

Our analysis is unique in that we provide a theoretical underpinning to the growing body of empirical evidence on corporate open source participation. We complement previous literature in this field with tools and insights of New Institutional Economics. In a broader context, this chapter links the literature of technology outsourcing to the phenomenon of corporate open source software.

Additionally, we present a novel approach to measure incomplete contracts. The chapter has four parts. The next section presents the theory of incomplete contracts and the relation with corporate open source. Next, we introduce the data used for our preliminary results. Then, we establish an econometric model for the SourceForge data. Lastly, we discuss the results for both the regressions and the survey.

3.2 Background

3.2.1 Incomplete Contracts

We use a simplified setup to represent corporate open source software. In this setup, we have a firm and a development community who provides an open source application. The firm needs a computer application to perform a certain task. If the firm decides to use the open source application, a transaction takes place between the firm and the development community. The application has a given set of services in terms of features. The development community in turn adds new features and improves existing functionality to the application.

Unlike other commercial transactions, the firm does not sign a legal contract for the open source application. The terms of trade between firm and application are laid out in the open source license. If the firm needs additional features or improved functionality, e.g. to accommodate new hardware, it has to interact with the development community. The interactions between firm and community can take the form of bug reports, support or feature requests, but can also be posts on mailing lists, submissions to fora or direct contacts with developers.

A bug report for instance is a short message to a bug tracking system that contains information about what in the application does not work and with which computer hardware and operating system the malfunction has been experienced. Here is an actual bug report from the JBoss project on Sourceforge.net:

```
''When I attempt to execute 'build all' on the latest source, using my w2k machine,
I get the following error.

BUILD FAILED java.lang.OutOfMemoryError

I have set ANT_OPTS=-Xmx640m.

Even the 'build clobber' command requires 60 MB of memory.  Does this make sense?
Isn't clobber just deleting a bunch of directories?

My latest attempt to 'build all' topped out at 90 MB.

Could someone please run this [program] through a profiler?  There's probably a
memory leak in there somewhere.  Given the nature of the task, it's hard for me
to believe that it really requires 90+ MB.

Are there any workarounds for this?''
```
[1]

Messages like the one above are then posted on the web site of the development community to which other users and developers can respond.

These messages are requests to subcontract development work to the community. Even though the open source license is not a formal contract (Laurent, 2004), it defines the terms under which interactions between the firm and the development community take place. The license sets only the framework for mutual collaboration. This means that it defines the terms of modification and redistribution, but waives all warranty claims and

[1] Here is a rough translation for the less technology savvy. The person who submitted the bug report apparently has a problem with creating the application from the source code with the given option of 'build all'. He uses the Microsoft Windows 2000 operating system and receives a message from the program that he does not have enough main memory. After explaining his problem, he asks the development community to run a program analysis tool, the so-called profiler, on the source code and suggests that there is a programming glitch in the source code somewhere.

responsibility for errors in the software (Rosen, 2004). In this regard, the open source license is indeed an incomplete contract (McGovan, 2001). It only defines a minimal set of conditions for the transaction.

Contracts are incomplete when they do not encompass all possible contingencies of a transaction (Williamson, 1967). The aim of a contract is to outline the terms of trade; the conditions under which a transaction takes place. Especially for complex contracts these conditions cannot include all possible future circumstances. If an unforeseen circumstance occurs, the incentives for the trading parties may change and they may deviate from the contract after signing it. This opportunistic behavior creates uncertainty about the contract *ex ante*. Why draw up a contract which the other party will not honor anyway? Knowing about the potential opportunism, trading partners might not engage in transactions in the first place.

To overcome contractual incompleteness and thus to enable the transaction, trading partners establish additional governance arrangements. These arrangements can range from full integration (Grossman and Hart, 1986), long-term contracts (Joskow, 1985), partial ownership agreements (Hennart, 1988), to off-setting relationship-specific investments (Heide and John, 1988). The firm thus imposes control over its transactions with a combination of contract and governance structures. Economic theory asserts that the trading partners will find the best arrangement to reduce the risk of opportunism (Shelanski and Klein, 1995). This means that all transactions we can observe already have sufficient governance structures in place.

When the firm decides to use an open source application, it faces also contractual incompleteness. As mentioned before, development communities supply new features, bug fixes and support services. All interactions between the development community and the firm are voluntary, however. Therefore, the firm does not know whether and when the development community treats its submissions and requests. The company cannot impose any conditions on the open source developers about the priorities of its requests.

The firm can reduce the incompleteness of the contract by setting up additional governance arrangements. Contributing in the development process of the open source community can be such an arrangement. Assigning a paid programmer to develop in an open source project is similar to partial vertical integration. The programmer participates in the development effort of an open source community and establishes a degree of control over the transaction between the firm and the community. He becomes the interlocutor for the firm inside the development community. In environments that change quickly and hence render the incompleteness of the contract more important, firms may need to impose more control over the development process and thus assign more developers to a project. Corporate contribution is a flexible way of vertically integrating open source software development. Doing so permits the firm to reduce the contractual incompleteness.

In New Institutional Economics, the attributes of the transaction play an important role in determining the incompleteness of the contract and create contractual hazards (Williamson, 1991). These attributes can comprise measurement difficulties and the frequency of repeated transactions, but also relationship-specific assets or weaknesses in the institutional environment. We can find these contractual hazards in the corporate open source software also.

Measurement difficulties and the frequency of repeated transactions are key elements in contractual hazards (Shelanski and Klein, 1995; Williamson, 1996). Difficulties in measuring the outcome of a transaction can lead to a principal-agent problem. When the outcome of a transaction cannot be fully ascertained, one trading partner may have an incentive to shirk the contract. In turn, the other partner may be unsure about the quality of the traded good and refrain from the transaction entirely. The frequency at which transactions take place plays an important role in determining contractual incompleteness. Few transactions over long periods of time increase the risk of uncertainty and the problem of incomplete contracts is more pronounced. The possibility of changing circumstances is high and, as unforeseen contigencies occur, contracts adjust only slowly. Moreover, with little repeated trading, partners do not face a reputational effect in contracting. One-off deviance cannot be punished easily and uncertainty about a transaction increases.

Relationship-specific investments (Klein et al., 1978; Joskow, 1985) can render the incompleteness of a contract more prevalent. When a firm has to invest in relationship-specific assets prior to a transaction, unforeseen contingencies or even a failure of the transaction is costly for the investing party. Asset specificity occurs when there is little salvage value to an investment outside the transaction. This creates a holdup problem for the partner who paid for the machine: Due to his investment, he has a weak bargaining position in case the contract is renegotiated. Again, knowing this beforehand, the partner might not be willing to invest in the relationship-specific asset in the first place and the transaction fails.

Embedded systems using open source software are such specific assets. These systems are electronic devices for specific purposes or built as fixed hardware-software bundles, such as GPS receivers, medical devices or cellphones. The cost to adapt the software to the firm's hardware and to meet the functional requirements can be significant. The specificity of the embedded system arises through the modifications to the open source software as well as the design of the electronic hardware. Corroborating our contention, embedded systems are prominent examples of corporate open source participation (Henkel, 2006).

One well-documented example of an embedded system is the Maemo software platform and Nokia's smartphones. Maemo is an operating system for cellphones which is based on open source software components (Jaaksi, 2006). Using open source software, Nokia still incurs cost to modify the software and adapt the hardware. As Jaaksi (2009), Nokia's vice-president of open source operations, points out "[i]n addition of getting the most significant parts of the code from community projects, the Nokia Maemo team has a huge job to develop, finalize, optimize, fine tune, test, and integrae [sic] the devices into ready packages." These investments in the embedded platform are made prior to marketing the actual smartphones. Future transactions between Nokia and open source applications run the danger of a holdup due to the specificity of the Maemo platform and the handheld devices. To reduce this risk, Nokia shares not only its modifications, but also participates in the general development of the open source applications.

Lastly, weak institutational environments can reinforce the incompleteness of contracts. The strength of the intellectual property right regime is important in the transaction between the firm and the open source application. In case the firm already has additional governance arrangements in place, for example a corporate developer, the type of open source license influences the contractual incompleteness. The corporate developer works in the development community and contributes source code to the application. These contributions are investments in the open source application. The firm wants to appropriate its investment and protect its intellectual property in the open source application (Dahlander and Magnusson, 2005; Fosturi et al., 2008).

The appropriation of intellectual property in open source software is difficult. The type of the open source license plays an important role. Some types of open source licenses allow the use of the source code in proprietary software and thus facilitate appropriation, others do not. Scholars call the former less-restrictive, or academic licenses, and the latter restrictive, or strong copyleft licenses. According to which type of open source license the open source application runs, the firm needs to establish more governance arrangements, perhaps vertically integrating the open source application and obtaining the entire copyright over the application's source code.

This gives the firm another way to impose control over its open source transactions, apart from corporate contribution. The firm can retain the copyright of the source code and oblige other contributors to transfer the copyright of the source code to the firm. This method gives the firm legal leverage to enforce its control over the development community. It is a common practice for corporate-led open source applications. MySQL, Mozilla and Microsoft are typical examples of this practice. We note that the success of this method depends crucially on the willingness of the open source community to accept these constraints and trust the company in its open source comittment (Shah, 2006).

3.3 Data

We use a twofold approach to the analysis of corporate open source software in light of incomplete contract theory. First, we estimate a reduced form model with a large data set. Our regression relates the share of corporate developers with proxies for contractual hazards and incompleteness. Second, we review the responses of 60 project maintainers in an exploratory survey. The responses and comments that we obtained in this survey shed some light on the interaction between development community and firm. We find that the two analyses give converging pieces of evidence on the validity of incomplete contract theory in corporate open source software.

3.3.1 Quantitative Data

We use data of development communities, or so-called projects, on SourceForge. SourceForge (SF) is an Internet platform that acts as an intermediary between project initiators, developers and users of open source software. Subscription to SF is free and only needed to participate in any phase of project development, i.e. for fixing a bug, contributing code or authorizing another release. SF offers the necessary infrastructure to maintain developer communities, the bandwidth for downloads as well as facilities to manage mailing lists and search facilities for users. We track 2,643 projects over a time period of 28 months from February 2005 until June 2007 (N = 74,004). Table 3.1 presents a list of all variables.

We retrace the possible factors that affect a firm's perception of contractual incompleteness and lead it to partially integrate the development of an open source application. According to our hypothesis, a firm would choose to contribute if the expected probability of having its request fixed by the community is low. To approximate this probability, we use characteristics of the development community about the lengths of time taken to treat submissions and the number of treated submissions.[2]

[2]On a technical note, the uncertainty disappears, if a firm knows the exact probability distribution. So, we have to assume some form of imperfect information on the part of the company. To mimic this, we use variables that are observable by the firm, but that do not assume a given probability distribution

Table 3.1: Summary Statistics (N=74,004)

Variable	Description	Mean	S.D.	Min	Max
cprate	Corporate participation rate	0.17	0.33	0	1
dur_FRsdmth	Duration variability (feature requests), months	10.46	8.64	0	48.5547
dur_FRmeanmth	Duration mean (feature requests), months	33.91	16.77	0.78333	91.6
dur_SRsdmth	Duration variability (support requests), months	8.50	7.78	0	48.5547
dur_SRmeanmth	Duration mean (support requests), months	35.93	16.82	0.26667	91.6
dur_BRsdmth	Duration variability (bug reports), months	12.55	9.44	0	45.2548
dur_BRmeanmth	Duration mean (bug reports), months	29.75	17.04	0.3	91.6
r_FRfixed	Number of fixed feature requests per developer	1.50	6.19	0	165
r_SRfixed	Number of fixed support requests per developer	0.66	5.82	0	243
r_BRfixed	Number of fixed bug reports per developer	5.54	22.70	0	568
licGPL	Strong copyleft licenses	0.68	0.47	0	1
licAca	Academic licenses	0.18	0.38	0	1
licLGPL	Weak copyleft licenses	0.10	0.30	0	1
licOth	Other licenses	0.01	0.11	0	1
statPlan	Planning stage	0.02	0.15	0	1
statPreA	Pre-Alpha stage	0.06	0.24	0	1
statAlph	Alpha stage	0.15	0.36	0	1
statBeta	Beta stage	0.32	0.47	0	1
statProd	Production stage	0.38	0.49	0	1
statMatu	Mature stage	0.03	0.17	0	1
statInac	Inactive	0.01	0.12	0	1
audEnd	End-user audience	0.32	0.47	0	1
audDev	Developer audience	0.52	0.50	0	1
audSys	System administrator and other audiences	0.12	0.33	0	1
osPos	Posix OS	0.22	0.41	0	1
osWin	Microsoft OS	0.13	0.34	0	1
osMid	Middleware	0.32	0.47	0	1
osOth	Embedded OS	0.15	0.35	0	1

We use the percentage of corporate developers with respect to all contributors in a development community as a measure of firms' involvement in a project. This corporate participation rate (cprate), see Table 3.1, lies well below the findings of other studies on corporate contributions Ghosh et al. (2002); Lakhani and Wolf (2005); Hammond et al. (2009). However, previous empirical research on SF notes that a large share of projects are abandoned or have only one developer (Howison and Crowston, 2004). Bearing the strong heterogeneity of SF projects in mind, it is also not surprising that participation rates are distributed very unevenly. Few projects attract a lot of corporate attention, whereas a large share has no paid developer at all.

Our premise is that the measures for contractual incompleteness and contractual hazards influence the firm's choice to contribute in the open source community and thus to establish additional governance. By contrast, we assume that corporate contribution does not affect our measures of contractual hazard in return. This means that we can observe our measure of contractual incompleteness, even though there is corporate participation.

This might seem paradoxical because one would normally expect contractual incompleteness to diminish when the firm establishes additional governance. However, we believe that the firm deals only with the corporate developer witihin the open source community. As mentioned before, the corporate developer acts as an interlocutor between firm and community.

Our proxies for contractual incompleteness and hazards comprise all requests to the development community, not only the corporate ones. Therefore, the firm may perceive a mitigation in contractual incompleteness, while we observe no reduction in our proxies. Put differently, those corporate contributors we observe in the development community specifically work on the requests of the firm and do not affect the overall functioning of the development community.

We look at three types of submissions. In our data sample, users or developers can

about the treatment of submissions. This prevents us from using more sophisticated econometric methods to measure transactional uncertainty.

submit either a bug report, file a support request or ask for a particular feature. These three categories differ in terms of the overall submissions received, of the development effort needed and also of the value to the development community. We measure the lapsed time between submission and treatment as well as the number of treated submissions. Both variables are proxies for the probability of having one's submission treated. The first variable considers more closely the time dimension of a transaction. Timeliness is important for corporate users. We can extend the duration variable a little further. We calculate the average duration of a submission as well as its standard deviation. These variables give us an indication about the average time until a submission is treated. For the other measure, we compute the number of treated submissions per developer for each open source community. The number of treated submissions indicates the frequency of transactions taking place for each community and using the per-developer average allows us to compare communities of different sizes.

The proxies for contractual incompleteness are (i) the time it takes a community to resolve a submission and (ii) the average number of submissions per developer each community treats. Submissions in SF, such as bug reports (BR), feature requests (FR) or support requests (SR), can come from developers or users. For each of the three main categories, we calculate the mean (dur_XXmeanmth) and standard deviation (dur_XXsdmth) of the duration of solving the submission. The duration goes from the date of submission to the date of reporting it fixed. We censor the data at the last month in case the submission has not been fixed during the observed time period. Our data set includes submissions from before our observation period. In fact, 80% of them were submitted before February 2005, the first month we track data on project properties. We can see that the average lifespans of bug reports, feature requests or support requests are around 30 months.

The other variables we use to measure contractual hazards are the number of treated feature requests per developer (r_FRfixed), the number of closed support requests per developer (r_SRfixed) and the number of fixed bug reports per developer (r_BRfixed). In contrast to the lifespan of a submission, we here look at the frequency of the transactions between users and community. These variables capture the average resolution rate per developer in each community. The standard deviation shows that the average resolution rates vary a lot across development communities.

SF projects are grouped into seven development stages: planning, pre-alpha, alpha, beta, production, mature or inactive. These are indicators about a project's life-cycle and represent the stability and number of features of a project. The pivotal stage is generally beta status in which, in contrast to earlier stages, the software runs smoothly and encounters relatively few crashes. The head of project chooses the development stage. There is one particularity with SF projects as there also exists an inactive development stage, in which development has basically ceased. Our sample contains a considerable number of projects in beta or later development phases (71%). We also find that the share of inactive projects (1%) is smaller than in other studies on SF projects (Howison and Crowston, 2004).

Like previous research (Perens, 1999; Lerner and Tirole, 2002), we distinguish open source licenses into three categories: academic, weak copyleft and strong copyleft licenses. These categories follow license's restrictions to use derivative work in proprietary software and their obligations to licensees to distribute modified software under the same license. Along these two axes, academic licenses allow the combination of derivative work in proprietary applications and do not require the same license for any modified version. weak copyleft licenses, in turn, allow the combination of modified software in proprietary applications, but oblige licensees to publish derivative work under the same license. strong copyleft restricts the combination of derivative work with proprietary applications and requires the continuation of the same license for any work that is based on the original source code. The projects in our sample run to 69% under strong copyleft, to 17% under academic and to 10% under weak copyleft licenses. The category "Other licenses" encompasses all projects that are signed with an older, now defunct, license.

We also have categories for the audiences the project is intended for and the operating system for which it is developed. SF lists 19 intended audiences among which are a variety of general fields, such as science, government or end-user. We adopt a categorization used by Lerner and Tirole (2002): end-users, developers and system administrators. We find that 53% of the projects are intended for developers, 33% for end-users and 12% for system administrators. This slightly skewed distribution towards software development can be expected from a user-driven development model. The distribution of operating systems is more interesting. Here the most popular system is middleware (32%), followed by Posix (23%), embedded systems (15%) and Microsoft-based operating systems (13%). Middleware encompasses all projects that run independently of the operating system. It acts as an interpreter between the application and the operation system. The Posix category includes all varieties of Unixes, such as Linux, AIX or Solaris.

There are two main sources of measurement errors within the data. First, there is a sample selection bias in the SF data for small and medium-sized projects. Previous studies (Howison and Crowston, 2004; Lerner and Tirole, 2005; David and Rullani, 2006) show that SF contains a large number of small or inactive projects. Without a comparable data set with projects outside a repository, our result cannot be verified for statements on OSS in general. A possible other consequence is that we underestimate project activity in the actual SF project population. The second source of measurement errors is more important for this study. We likely face type I and type II errors on corporate affiliations that are due to our data collection method. Corporate developers might use their work email addresses for private purposes. Thus, even though we find a corporate affiliation, there is in fact no coordinated firm strategy on OSS. The potential type II error is that corporate developers could use their private email addresses for work related projects. Without interviewing each developer, there is no possibility of knowing how strong these two effects are.

Table 3.2 repeats the predictions based on the hypotheses we established with regards to corporate open source software. We expect that the variables for the treatment duration have a positive impact on the expected participation rate. The longer it takes and the more distributed the time-span between submission and fixing becomes, the stronger

Table 3.2: Predictions

Corporate participation rate is high...	Variables	Sign
... for communities with high contractual uncertainty.		
	dur_FRsdmth	+
	dur_SRsdmth	+
	dur_BRsdmth	+
	dur_FRmeanmth	+
	dur_SRmeanmth	+
	dur_BRmeanmth	+
	r_FRfixed	-
	r_SRfixed	-
	r_BRfixed	-

is the contractual uncertainty and the more likely it is that a firm assigns a developer to the community. Along the same lines, the higher the number of treated requests per developers is, the higher is the frequency of transactions and the smaller is the contractual hazard.

3.3.1.1 Econometric Model

We want to regress the share of corporate developers among all developers on a set of project characteristics and measures of contractual incompleteness. The corporate participation rate is bounded between 0 and 1 with a positive probability of being at each limit. Ordinary least squares has the downside that it cannot ensure that the predicted values lie within the bound *per se*, similarly to the well-known linear probability model. Tobit estimation, on the other hand, takes into account these bounds, but treats them as truncations and thus does not include them in the calculation.

Papke and Wooldridge (1996) propose an estimation method that takes into account the special nature of the explained variable and proves to be robust to misspecification. They estimate the conditional mean of the explained value with the nonlinear function $G(.)$, which satisfies $0 \leq G(.) \leq 1$, and compute the parameters with a quasi-maximum likelihood estimation. In most applications $G(.)$ is a cumulative distribution function, such as a logit, a probit or a loglog function.[3] The conditional mean of the corporate participation rate on the explanatory variables is:

$$E(y|x) = G(x\beta) \tag{3.1}$$

We then obtain the estimates with a Bernoulli log-likelihood function:

$$LL(\beta) = y\, log[G(x\beta)] + (1 + y)\, log[1 - G(x\beta)] \tag{3.2}$$

Setting a functional form for $G(.)$, we derive the influence of each explanatory variable and the predicted values for corporate participation rates. Following Papke and Wooldridge (1996), we use the logit function for $G(.)$. Since it is a nonlinear function, the effects of changes in one explanatory variable on the expected value of corporate participation have to be simulated for specific values of all explanatory variables, $\bar{x}$:

$$\frac{\partial E(y|x)}{\partial x_i} = \beta_i G'(\bar{x}\beta) = \beta_i G(\bar{x}\beta)\,[1 - G(\bar{x}\beta)] \tag{3.3}$$

Note that we estimate the coefficient as a cross section instead of a panel and adjust the standard errors for clustered data, in this case each project. Two properties of the data sample justify this method. First, the data varies very slightly over time. For example, overall participation rates change for only 55 of the 74,004 observations. Running regressions on the actual cross sections for each time period show that there is no significant difference between the two approaches (see table C.1). Secondly, we have to adjust standard errors for heteroskedacity (Papke and Wooldridge, 1996) and can do so using the clustered robust standard errors, which in this case are similar (see table C.1).

[3]An added benefit of this estimation method is that it can be easily done using standard econometric software.

3.3.2 Exploratory Survey

Between June and July 2009, we conducted an email survey on 7,914 project maintainers and received 60 valid responses.[4] Although the response rate is quite low at 0.76% compared to Henkel (2006), it is inline with research using a similar methodology (Haaland et al., 2009). The list of project maintainers comes from the SourceForge database at Notre Dame University (Gao et al., 2007). We contacted them in batches of 50 over a time period of several days and recontacted unanswered emails after a few weeks.

The survey contains four questions that aim at the treatment of user requests by the development community. The questions touch upon the treatment of generic user requests, the treatment of corporate requests as well as the use of a ranking system to signal priorities or personal communication to motivate voluntary contributions. We focus on user requests because we believe they are good indicators for the interaction between development community and firm. Note that the survey does not look at the transaction between the open source application and the firm, but rather at the attitude within the development community towards corporate requests. Our contention is that the less interested the development community is to fix corporate requests, the stronger weighs the contractual incompleteness on the firm and to more incentive it has to contribute in the development.

Figure 3.1 shows the responses to question 1. Multiple answers are possible so that the overall number of responses does not necessarily coincide with the number of respondents. Project maintainers identify difficulty and low interest as the main reasons for slow responses to user requests. One respondent states that ”[a]s soon as a project reaches a certain size (typically attained by big user-oriented applications like a word processor for instance), it becomes very hard to recruit developers. I believe the reason is simply that in the case of volunteer collaboration, you can only devote parts of your free time to the project.”

Figure 3.2 depicts the responses to question 2. Project administrators believe that

[4]Appendix B shows the entire email message with the survey.

little interest from voluntary contributors is the prevalent reason for corporate requests to take long. The lack of interest may be linked to the specificity of corporate requests. One administrator believes that “[...] businesses often want a much more complete and personally tailored solution, as opposed to developers who can handle much of the work themselves after you give them a small amount of assistance.” Along the same lines, another respondent states ”[...] requested changes sometimes enhance the project itself very little, and instead are geared towards meeting the requesting business’s goals instead.” It appears that the firm faces considerable contractual hazard in trading with the open source application and in turn dealing with the development community. The disinterest of the development community in corporate requests renders repeated subcontracting of corporate development work unlikely or difficult. To reduce the contractual hazard, the firm needs to establish additional governance arrangements.

Figure 3.3 shows the responses to question 4.[5] To accelerate the treatment of requests, project maintainers rely frequently on their own effort and only occasionally on others. This confirms a property of the development community that Shah (2006) has pointed out before: The project maintainer has limited executive authority in his project. He cannot command voluntary contributors to work on particular requests. If they do not choose themselves, he often has to work on the request himself. This finding has an important implication for the firm engaging with the development community. To establish governance, it is not enough for the firm to win over the project administrator or, in a more extreme case, to replace him with a paid developer. The firm has to take into account the development community to keep the external development going and create additional governance arrangements.

The discussion of our survey shows that (i) corporate requests may incite less interest from the development community and (ii) the project administrator has little influence on the behavior of other contributors. The first point helps to interpret the results of the

[5]We skip question 3 because it overlaps with question 4 and adds little additional insight for our analysis at hand. The responses can be seen in figure B.1 in Appendix B.

Figure 3.1: Do you think requests in general take long, because they...

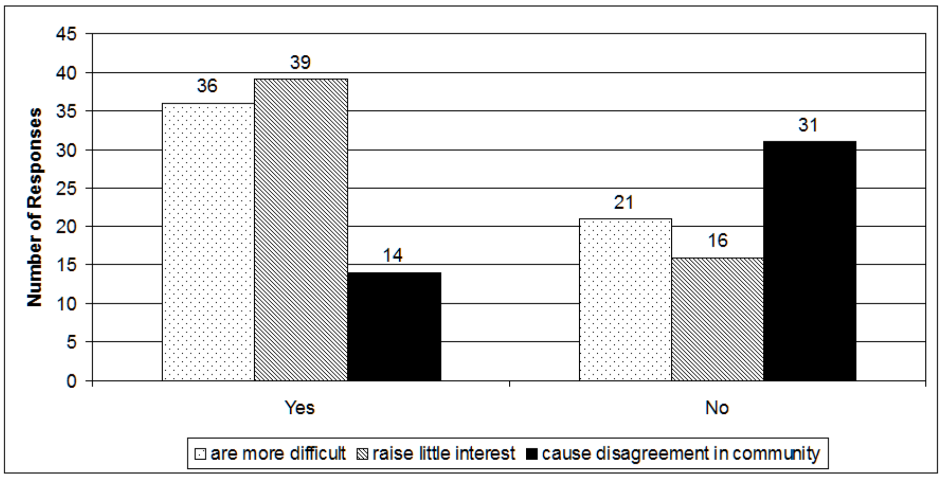


Figure 3.2: Do you think requests by firms take long, because they...

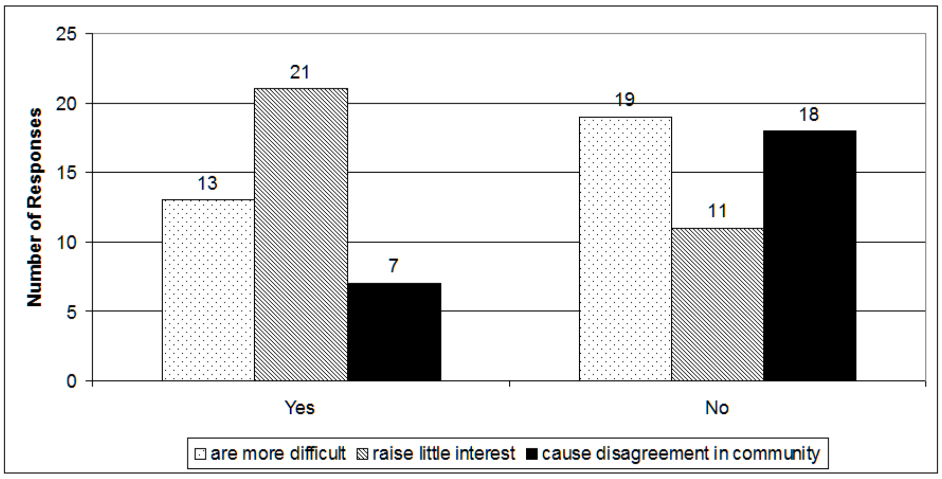
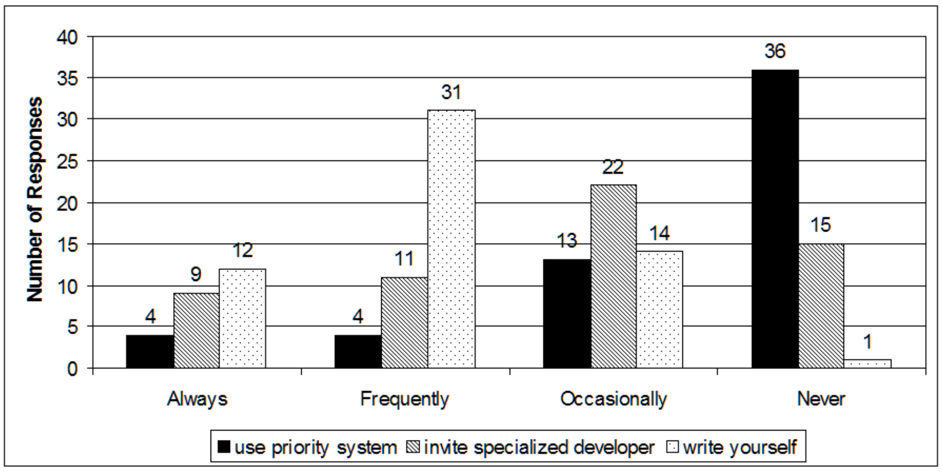


Figure 3.3: To accelerate requests do you...



econometric analysis in the following section, when we look at the mean treatment time and its variability of different types of requests. The second point shows that the attitude of the open source developers towards the firm is an important factor in the transaction between the firm and the open source application.

3.4 Discussion

Table 3.3 presents the estimates for the regression of corporate participation rates on the transaction uncertainty variables and the control variables. We add stepwise more control variables to the equation. None of the added control variables in columns 2 to 4 are significant. The joint hypothesis tests on the control variables on status, audience and operating systems are all statistically insignificant. For model 4, the respective χ^2 values (p-values) are 4.14 (0.66), 3.65 (0.16) and 2.21 (0.53). Also, the RESET tests show that the functional form is not misspecified, e.g. that the population model includes squared terms of mean duration. In fact, the simplest model is the preferred one. In the following analysis, we will focus on model 1 and use its parameter values to compute the marginal effects.

We see that indeed the duration variability for treating feature requests (dur_FRsdmth) and the respective mean duration (dur_FRmeanmth) are statistically significant and positive. This suggests that the more time it takes to fix a request or the more variable the treatment of requests is the higher is the expected corporate participation rate. Also, the average number of feature requests fixed per developer (r_FRfixed) has a, strongly significant, negative effect on corporate participation. Increasing the number of treated feature requests per developer thus translates into lower expected corporate participation rates. These results lend support to the notion that firms contribute to reduce the incompleteness of their open source transaction.

Combining these results with the findings of our exploratory survey allows us to draw a more complete picture. To wit, our respondents state that requests in general take long because there are programming difficulties or because there is little interest by the development community. Project maintainers also contend that corporate requests more often encounter little interest than requests in general. This leads us to believe that the firm experiences strong contractual incompleteness and contractual hazard due to disinterest when there is a long mean duration of requests and larger variability.

The results for r_BRfixed and r_SRfixed in Table 3.3 are counter-intuitive. The average number of bugs fixed per developer (r_BRfixed) has a positive effect on the expected participation rate. The average number of support requests per developer (r_SRfixed), although not significant at the 5% level, points in the same direction. We are likely to conflate two effects here. On the one hand, there is an uncertainty effect, the longer it takes or the fewer requests are resolved, the higher is the contractual hazard that one's own submissions will not be treated. On the other hand, there is a performance effect. Better performing projects are more likely to fix bugs and address feature requests than less performing ones and could therefore also attract more attention from firms. If all three variables, r_FRfixed, r_BRfixed and r_SRfixed capture the same effect, this can create a problem in the results. The variables are likely correlated and thus adding the three can bias the coefficients of all three. The problem becomes clear when we attempt to interpret the results with respect to our predictions. Does a positive coefficient for the average number of fixed bugs mean for instance that we do not find supporting evidence for our transaction cost hypothesis or does this simply capture the performance effect?

One possible solution is to make use of the omitted variable bias. Assuming that the three variables represent the same effect and that they thus are correlated, we can run several regressions including all variables except for this uncertainty measure. We then observe whether the coefficients of the remaining variables change significantly. If they do, we know that the three variables represent the same underlying effect and are correlated in some form. The change in the remaining coefficients shows that these take up part of the effect of the omitted variables. In contrast, if the coefficients do not change signs, it lends support to the notion that in fact they measure two separate effects altogether.

Hence, if the performance effect is prevalent among the set of omitted variable and the remaining variables, the estimates of the remaining variables will change signs. If the different variables measure two separate effects, they do not change.

Table 3.4 shows the estimates for the complete model (column 1) and the six regressions with different sets of uncertainty measures omitted. We see that, although the duration variables for feature requests vary, they remain positive across all steps (columns 3 - 7). The duration variables for support requests and bug reports fluctuate around zero for all iterations. This supports the finding that they are statistically insignificant, with the exception of the mean treatment duration of support request in column 5. More interesting for our purposes are the coefficients for average treated requests per developer. The coefficients for the average number of fixed feature requests remain negative in all columns. The coefficients for the average number of fixed bug reports is positive across all columns and the the coefficients for r_SRfixed remains also relatively stable.

What can we take away from the discussion of the last two paragraphs and the analysis in table 3.4? Assuming the potential direction of the omitted variable bias, we are able to show that the signs of the parameters for contractual incompleteness do not change significantly as we leave out different variables in the regressions. *This lends support to our assertion that (i) dur_FRsdmth and dur_FRmeanmth capture part of the contractual incompleteness, (ii) r_FRfixed can be interpreted as a measure of transactional frequency, or more appropriately of subcontracting frequency, and (iii) r_BRfixed can be seen as a performance measure for the development community.*

As we include more control variables, the coefficients stay stable. In all, we find supporting evidence for the hypothesis that less-restrictive licenses induce higher corporate participation. The coefficients for the intended audiences suggest that expected corporate participation rates are higher for projects directed towards developers compared to projects intended for end-users, but the estimate is insignificant at the 5% level. Lastly, we find no significant effect of operating systems on corporate participation rates.

Figure 3.4 presents the marginal effects for the variables dur_FRsdmth and dur_FRmeanmth

Table 3.3: Regression on Corporate Participation

	(1)	(2)	(3)	(4)
dur_FRsdmth	0.0262*	0.0260*	0.0257*	0.0257*
	(0.0105)	(0.0105)	(0.0106)	(0.0106)
dur_FRmeanmth	0.0340+	0.0340+	0.0341+	0.0340+
	(0.0114)	(0.0115)	(0.0114)	(0.0114)
dur_SRsdmth	-0.0080	-0.0096	-0.0102	-0.0104
	(0.0108)	(0.0109)	(0.0109)	(0.0109)
dur_SRmeanmth	-0.0184	-0.0188	-0.0193	-0.0192
	(0.0111)	(0.0111)	(0.0111)	(0.0111)
dur_BRsdmth	0.0044	0.0054	0.0052	0.0052
	(0.0084)	(0.0084)	(0.0084)	(0.0084)
dur_BRmeanmth	-0.0032	-0.0023	-0.0023	-0.0024
	(0.0063)	(0.0063)	(0.0063)	(0.0063)
r_FRfixed	-0.0419*	-0.0427*	-0.0420*	-0.0421*
	(0.0172)	(0.0174)	(0.0168)	(0.0168)
r_SRfixed	0.0062	0.0059	0.0055	0.0056
	(0.0068)	(0.0069)	(0.0069)	(0.0069)
r_BRfixed	0.0089+	0.0089+	0.0087+	0.0087+
	(0.0034)	(0.0034)	(0.0033)	(0.0033)
licAca	0.3484+	0.3542+	0.3325+	0.3407+
	(0.1089)	(0.1094)	(0.1109)	(0.1109)
licLGPL	0.1612	0.1542	0.1096	0.1212
	(0.1415)	(0.1406)	(0.1424)	(0.1435)
licOth	-0.0258	-0.0326	-0.0225	-0.0140
	(0.3971)	(0.3945)	(0.4018)	(0.4015)
statPlan		-0.0501	-0.0479	-0.0421
		(0.3066)	(0.3084)	(0.3087)
statPreA		-0.0963	-0.0961	-0.0916
		(0.1980)	(0.1984)	(0.1987)
statAlph		-0.2759	-0.2810	-0.2798
		(0.1435)	(0.1437)	(0.1436)
statProd		-0.0230	-0.0256	-0.0230
		(0.1068)	(0.1066)	(0.1068)
statMatu		0.0332	0.0256	0.0207
		(0.2395)	(0.2403)	(0.2416)
statInac		-0.0573	-0.0574	-0.0445
		(0.3896)	(0.3930)	(0.3925)
audDev			0.0937	0.1065
			(0.0967)	(0.0981)
audSys			-0.1824	-0.1698
			(0.1577)	(0.1588)
osWin				-0.0285
				(0.0991)
osMid				-0.0449
				(0.0979)
osOth				0.0973
				(0.0767)
constant	-2.3488+	-2.3014+	-2.2972+	-2.2990+
	(0.1370)	(0.1512)	(0.1600)	(0.1625)
N	74,004	74,004	74,004	74,004
RESET	0.436+	0.693+	0.460+	0.415+

Standard errors in parentheses * p<.05 + p<.01

Table 3.4: Control Regressions on Corporate Participation

	(1)	(2)	(3)	(4)	(5)	(6)	(7)
dur_FRsdmth	0.0262*		0.0168*	0.0274+	0.0316+	0.0276+	0.0307+
	(0.0105)		(0.0071)	(0.0101)	(0.0102)	(0.0104)	(0.0102)
dur_FRmeanmth	0.0340+		0.0185+	0.0319+	0.0469+	0.0363+	0.0438+
	(0.0114)		(0.0066)	(0.0108)	(0.0104)	(0.0113)	(0.0107)
dur_SRsdmth	-0.0080	0.0131		-0.0053	-0.0121	-0.0094	-0.0082
	(0.0108)	(0.0077)		(0.0103)	(0.0107)	(0.0108)	(0.0107)
dur_SRmeanmth	-0.0184	0.0091		-0.0183	-0.0264*	-0.0209	-0.0207
	(0.0111)	(0.0064)		(0.0108)	(0.0107)	(0.0109)	(0.0107)
dur_BRsdmth	0.0044	0.0102	0.0029		0.0010	0.0045	-0.0017
	(0.0084)	(0.0081)	(0.0080)		(0.0083)	(0.0084)	(0.0084)
dur_BRmeanmth	-0.0032	0.0031	-0.0054		-0.0074	-0.0031	-0.0109
	(0.0063)	(0.0059)	(0.0062)		(0.0061)	(0.0063)	(0.0061)
r_FRfixed	-0.0419*	-0.0600+	-0.0500+	-0.0440*		-0.0384*	-0.0115
	(0.0172)	(0.0179)	(0.0181)	(0.0171)		(0.0175)	(0.0094)
r_SRfixed	0.0062	0.0120	0.0102	0.0064	-0.0014		0.0040
	(0.0068)	(0.0065)	(0.0064)	(0.0068)	(0.0075)		(0.0064)
r_BRfixed	0.0089+	0.0113+	0.0095+	0.0095+	0.0032	0.0087*	
	(0.0034)	(0.0035)	(0.0035)	(0.0033)	(0.0018)	(0.0034)	
licAca	0.3484+	0.3403+	0.3463+	0.3479+	0.3620+	0.3509+	0.3618+
	(0.1089)	(0.1085)	(0.1085)	(0.1091)	(0.1083)	(0.1091)	(0.1084)
licLGPL	0.1612	0.1792	0.1691	0.1614	0.1586	0.1598	0.1586
	(0.1415)	(0.1413)	(0.1414)	(0.1412)	(0.1414)	(0.1414)	(0.1412)
licOth	-0.0258	-0.0641	-0.0321	-0.0354	0.0186	-0.0251	0.0251
	(0.3971)	(0.3975)	(0.3926)	(0.3975)	(0.4006)	(0.3972)	(0.4080)
constant	-2.3488+	-2.3362+	-2.3615+	-2.3542+	-2.3707+	-2.3436+	-2.3255+
	(0.1370)	(0.1368)	(0.1365)	(0.1369)	(0.1362)	(0.1363)	(0.1366)
N	74,004	74,004	74,004	74,004	74,004	74,004	74,004
Standard errors in parentheses	* p<.05	+ p<.01					

Figure 3.4: Marginal Effects 1

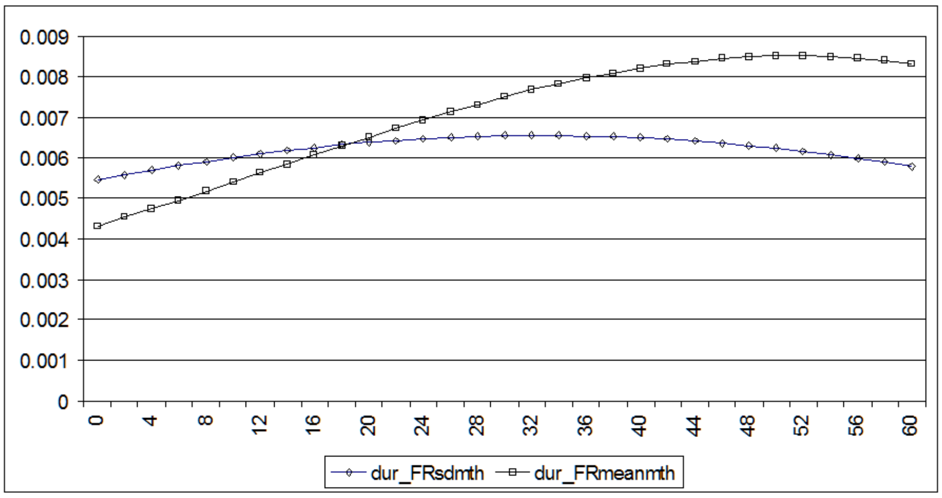
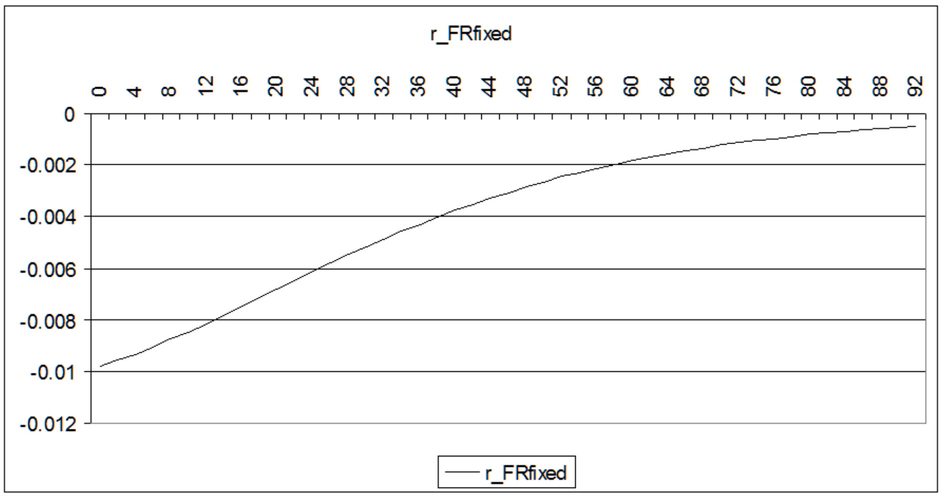


Figure 3.5: Marginal Effects 2



on the expected corporate participation rate for a project under an academic license. We have fixed all other variables on either their mean values or zero for statistically insignificant coefficients. For dur_FRsdmth the effect reaches a maximum at 33 months, after which the variability of time to treat feature requests has a positive, but diminishing, impact on the expected corporate participation rate. The marginal effect of dur_FRmeanmth reaches a maximum at 53 months at which date increases in the average time to treat feature requests increases the expected participation rate to a lesser extent. Figure 3.5 shows the marginal effect of the average number of fixed feature requests per developer, r_FRfixed. It decreases continuously and approaches zero.

Endogeneity is an issue in our analysis. It occurs because corporate participation likely affects the average number of treated feature requests per developer, the average duration for each request and its variability. This causes a bias in the estimated coefficients. The standard way to treat endogeneity is to instrument the variables. Unfortunately, we are unable to appropriately instrument all variables. We caution therefore that the coefficients are likely biased. The direction of the endogeneity bias depends on the impact of corporate participation on the treatment of requests. If communities with a lot of corporate developers respond more slowly to submissions and hence increase mean duration and variability, we will obtain an upward bias. In case, corporate developers positively affect the treatment of submissions, we have a downward bias and we obtain too negative estimates compared to the population coefficients. Lamastra (2009) explores the influence of corporate developers on the quality of the development of SF projects. She shows that, when corporate developers enter SF communities, code development increases and non-development activities decrease in terms of quality. This may indicate that high corporate participation rates positively affect the treatment of feature requests and bug reports and negatively influences those of support requests. We might observe too small an effect for the treatment of feature requests and bug reports and too large an effect on support requests. Considering Table 3.3, these findings confirm our results and corroborate our interpretation.

3.5 Concluding Remarks

The question on how to control open source projects is of paramount importance for corporate strategy in open source software. We attempt to shed light on some aspects of this issue. With regards to contribution, we find evidence that firms opt for projects with higher measures of contractual incompleteness. This indicates that firms establish additional governance arrangements in form of dedicated corporate contributors in open source communities. This type of vertical integration mitigates part of the risk of transacting with open source applications under an incomplete contract.

There are ample points of entry for future research on corporate contribution, control and open source software. First, the analysis of corporate control on open source software could be enhanced with adding firm characteristics and refining the measures for project control. Second, the strategic behavior of firms within open source projects merits further consideration. Do firms cooperate in common projects or do they drive other companies out of projects? Third, the interaction between private and corporate developers raises interesting questions on the development of individual incentives to contribute.

On a different note, we show that corporate open source software can be understood with the tools provided by the theory of incomplete contracts and transaction cost. This might open a way to investigate corporate provision of open source software in terms of technology outsourcing and open innovation. Indeed, the growing implication of firms in open source software may render it possible for social scientists to shed light on these concepts and gain more insights into the obstacles of external development and production of technology and innovation.

Conclusion

My thesis addresses three aspects of corporate involvement in open source software: the private provision of a public good, open innovation as well as technology outsourcing. We broach the first issue by way of a literature survey that focuses on the commercial viability of the provision of a public good. Accordingly, the first essay discusses the research on corporate use and provision of open source software. In the second essay, we turn to open innovation and look at the interaction between corporate and voluntary contributors in the production of an open source application. In particular, we investigate the potential productivity effects of this interaction as well as of knowledge spillovers which are often evoked as a great boon of open innovation. The third essay considers technology outsourcing in open source software and its governance issues. We contend that corporate contribution in open source software can be understood in terms of incomplete contracts and transaction cost.

What can we learn from open source software on the private provision of public goods? Despite its public good properties, open source software is amenable to corporate objectives. We present business models and ways of corporate involvement that are economically viable and engage volunteers at the same time. Moreover, we show what can happen to firms that directly compete with collectively-provided public goods.

Our insights can be useful for the analysis of sectors in which companies struggle with voluntarily-provided digital content, for instance newspapers, encyclopedias or television channels. News articles, editorials or encyclopedic entries are pieces of information that can be produced by a large number of individuals and redistributed easily at little cost. Prominent examples are Wikipedia, the blog aggregator Global Voices, the social networking site Twitter or the user-driven news site Reddit. These sites attract several million viewers a day and thus pose a threat to traditional media outlets. Although our examples are all cases with a global scale, our discussion works also, and perhaps even better, on a smaller scale between local media and local bloggers for instance.

The discussion about proprietary and open source software competition delineates several possible strategies for traditional publishing houses. We will mention only two of them here. They could invest more to obtain high quality content, to hire better journalists or to provide a more diverse set of news, therefore differentiating themselves from the volunteer-driven alternatives. This will ultimately be futile as it leads to an arms race between traditional media and online competitors that the traditional media cannot win. Volunteers will always be able to modify - some might say imitate - and to redistribute the news or editorials. Therefore the investment will not create a significant competitive advantage and the publishing house spends money for nothing.

Another strategy is to embrace the user-driven production method. By allowing readers and volunteers to contribute in the production of editorials, newspaper articles or content in general, traditional media companies may be able to kill two birds with one stone. They reduce production costs *without* decreasing the value to the consumer or the advertiser at the same time. Outsourcing part of the provision of content to volunteers allows traditional media houses to focus on other competitive advantages, for instance providing editorials, signalling reliable news or ameliorating the incentives to contribute. An example of a successful implementation is Google Knol. It is a free encyclopedia that pays contributors for submitting articles based on the number of views and sells online advertising.

In a digital network, a key aspect is the enforcement and protection of intellectual property. We have seen that in the case of open source software, the firm has several possibilities to deal with this issue. Publishers and commercial content providers, on the other hand, struggle to find similar means of protection.

Besides the provision of a public good, corporate involvement in open source software allows us to explore open innovation and its productivity benefits. Much has been written about the productivity gains in research made possible by including external actors. Scholars often attribute economies of scope and knowledge spillovers as the sources for these productivity gains. We attempt to shed light on this using the case of corporate involvement in open source software.

There is scarce empirical evidence about the productivity effect of rising corporate participation in open source development. As the number of corporate contributors inside software projects rises, the need to understand their effect on open source production becomes important. There are good reasons to believe that corporate participation may not improve productivity. Firms do not necessarily share the same objectives as voluntary contributors on the direction of the development process. Conflicts of interests might hampen software production and may ultimately lead to the forking of an open source project. Moreover, the work processes of corporate developers may not be readily applicable to the open source production model. Large companies have long delays in decision-making which might decrease a project's productivity as programming tasks are left undone in the meantime.

We find that there are indeed negative productivity effects due to the interaction of corporate developers with voluntary contributors. These negative effects might be due to the aforementioned friction and conflicts of interest. However, knowledge spillovers can mitigate parts of the loss in productivity and the predominance of corporate or voluntary contributors in a community might reduce the impact of this friction.

Can we generalize these findings to other cases of open innovation? Our results show that open innovation is not a panacea to boost research and technology production *per se*. To be successful, the firm needs to be aware of the caveats. Motivating the volunteers and aligning the objectives of the community with those of the firm is crucial. Even though this might seem obvious, cases of failed open source engagements show that firms do not always heed this advice. Moreover, the firm needs to learn to let go. To reap the benefits of its open source engagement, the firm might want to meet the development community halfway by contributing to the development. In the course of the open source engagement, the firm might re-evalutate its intellectual property.

In addition, corporate involvement in open source software might trigger a re-appraisal of the competitive advantages of a firm. Companies might see that the competitive edge is not the source code of a software, but the human component of software production. Open source software may augment the value of human capital and commoditize the actual piece of software. In this regard, open source software is a true process innovation with profound implications for ICT labor markets.

Related to the issue of productivity is the inter-firm collaboration inside open source communities. Even though it is widely acknowledged that firms collaborate in the development of open source software, researchers have yet to provide insights into how this collaboration works and what effect it has on software creation. In this sense, open source projects are similar to R&D joint ventures. Important questions in this respect are to what extent firms cooperate to establish common standards and how firms can appropriate their investment in the software development.

In the last essay, we touch upon the issue of contracting research and technology. The interaction between the firm and external suppliers of technology raises the question of governance and transactional uncertainty. In a world with an unknown future and limited resources, how can the firm ensure the fulfillment of its contract by the supplier? We believe that corporate open source software is an ideal case in point for the analysis of this question.

We argue that the firm establishes additional governance arrangements to ensure the completion of the transaction with the open source community. These arrangements can take different forms, for instance the acquisition of the ownership rights for the source code, source code hijacking or corporate contributions. Dedicating a developer to work on an open source project is a kind of partial vertical integration. The firm can scale the integration, i.e. dedicate more developers, depending on the perceived uncertainty of the transaction. Using this approach, we can provide a complementary explanation to the literature of corporate contributions in open source software.

Our analysis of incomplete contracts is useful for the evaluation of corporate participation in projects of crowd-sourced production. Crowd-sourcing is a concept that takes user-driven innovation and modularity in production and applies them to sectors outside the software industry. Its initiators hope to carry over the productivity benefits seen in open source software. As it stands, crowd-sourcing gains momentum in diverse fields, such as research, political activisim or Do-It-Yourself tinkering. Prominent examples are Nasa's Clickworkers, the UK's Conservatives "Make IT Better" program or Bug Labs' The Bug.

One of the advantages of including external developers in the production is the extent to which the firm can reduce its own production costs. Can the firm outsource enough to the crowd to make the production worthwhile? This is, above else, a question of creating sufficient incentives for external agents to contribute. These developers need to find some personal benefit or altruistic meaning in the production of the good. This said, there is a second part to this questions. How does the firm design the user contract so that the firm is not obliged to contribute too much?

The discussion of incomplete contracts and corporate contributions in open source sheds light on the governance structure between the firm and development community. Contractual hazards oblige the firm to contribute in the production of an open source application. For crowd-sourced production, this means that the firm faces a double-edged sword. On the one hand, it can create a very restrictive contract to reduce contractual hazards with fixed specifications, delivery deadlines or strict conditions of use. However, this may scare away external contributors and reduce the productivity benefits of crowd-sourcing. On the other hand, the firm can offer a contract similar to open source licenses, in which it delineates only the rudimentary rules of collaboration. Promoting contributions, this type of contract may mean that the firm needs to get involved too much to control the production.

We are interested in the commercial viability of corporate crowd-sourced engagements and the design of possible user contracts. How can the firm design a contract that entices users to contribute, while creating revenue for the firm? We do not have an answer for these questions, but we believe that our discussion about incomplete contracts and corporate contributions is a first step to understand this fascinating problem.

Appendix A

Source Code of Email Tracer Application

Listing A.1: Python Source Code for JET Application (jet.py)

```
#########################################
##
##              JET (Jan's Email Tracer)
##              version 3
##
##
#########################################

__author__ = 'Jan Eilhard'
__version__ = 'Version 3.0'
__date__ = 'Date: 24/11/2007'
__copyright__ = 'Copyright (c) 2007 Jan Eilhard'
__license__ = 'Open Software License'

import auxfunctions
import sys
from sys import argv
import os

def getConfig():
        """Ask questions on project name, configuration files &
           proxy settings
        """
        configuration = {}

        print "Welcome to the configuration of JET."
        print " What is the name of the project? ",
        try:
                configuration["project"] = sys.stdin.readline().
                   strip()
                print "Enter the filename which contains your
                   email addresses: ",
                filename = sys.stdin.readline().strip()
                configuration["emails"] = auxfunctions.checkTxt(
                   str(filename)) and os.path.join(filename) or
                   None
                print "Enter the names of the search categories (
                   single words only): ",
                configuration["search"] = {}.fromkeys(sys.stdin.
                   readline().strip().split(" "))

                for cat in configuration["search"].keys():
                        print "Please enter the filename
                           containing search terms for %s: " % cat
                        filename = sys.stdin.readline().strip()
                        configuration["search"][cat] =
                            auxfunctions.checkTxt(str(filename))
                            and os.path.join(filename) or None

                print "Do you have special proxy settings (Y/N)? "
                    ,
                configuration["proxy"] = (sys.stdin.readline().
                    strip() == "Y") and raw_input("Please enter
                    your proxy settings: ") or None
                print "\n".join(["%s = %s" % (k,v) for k,v in
                    configuration.items()])
                print "Is this configuration correct (Y/N)? ",
                (sys.stdin.readline().strip() == "Y") and
                    auxfunctions.createDir(str((configuration["
                    project"]))) or getConfig()
                print "Do you want to start the Internet search
                    right now (Y/N)? ",
                return (sys.stdin.readline().strip() == "Y") and
                    configuration or auxfunctions.saveFile(
                    configuration, "config")

        except:
                print "An error has occurred."

def startInterface(argv=None):
        """Dispatch either JET directly or run configuration
        """
        auxfunctions.clearScreen()
        try:
                configuration = auxfunctions.checkTxt(str(argv[2])
                    ) and auxfunctions.readFile(str(argv[2])) or
                    getConfig()
                startSearch(configuration)
        except:
                pass


def startSearch(configuration):
        """ startSearch(cofiguration): Gets configuration dict,
           prepares mails dictionary, then filterHTML
                & returns None (side effect: saves output file)
        """
        try:
                mails = auxfunctions.readFile(configuration["
                   emails"])
                print mails
                for key in configuration["search"].keys():
                        configuration["search"][key] = [elem for
                           elem in auxfunctions.readFile(
                           configuration["search"][key])]
                print configuration
                list = []
                for key in mails.keys():
                        list.append(key.partition("@")[2])
                mails = mails.fromkeys(["www." + elem for elem in
                   list])
                for key in mails.keys():
                        mails[key] = auxfunctions.searchSite(key,
                           configuration["search"])
        except:
                print "An error occurred in startSearch."
                return 1
```

```
if __name__ == "__main__" :
                startInterface(argv=None)
```

Listing A.2: Python Source Code for Auxillary Functions (auxfunctions.py)

```
# Created on November 19, 2007


def info():
        """Auxilliary functions for JET
        """


def clearScreen(numlines=100):
        """clearScreen(numlines=100): Clear the console.
        """
        print '\n' * numlines



def createDir(dirname):
        """createDir(dirname): Takes a string, creates directory
           for project
           & returns '1' or '0'
        """
        import os
        try:
                os.mkdir(os.path.join(os.getcwd(), dirname))
                return 1
        except:
                print "The name you entered for the project is
                   invalid. Please try again. \n"
                return 0



def getDict(list, delimiter="\t"):
        """getDict(list, delimiter): Gets list, converts into a
```

```
            dictionary using <delimiter>
        & returns a dictionary
        """
        keys = []
        values = []
        try:
                for elem in list:
                        dump = elem.partition(delimiter)
                        keys.append(dump[0].strip())
                        values.append(dump[2].strip())
                return dict(zip(keys,values))
        except:
                print "List to dictionary conversion failed."
                return 0

def readFile(filename, delimiter="\t"):
        """readFile(filename): Gets string <filename>, opens &
            reads data
            & returns dictionary
        """
        try:
                infile = open(filename, "rb")
                data = infile.read()
                infile.close()
                list = data.splitlines()
                datadict = getDict(list, delimiter)
                if datadict.has_key("search"):
                        datadict["search"] = datadict["search"].
                            strip("{")
                        datadict["search"] = datadict["search"].
                            strip("}")
                        datadict["search"] = datadict["search"].
                           replace('\'', '')
                        datadict["search"] = datadict["search"].
                           split(",")
                        datadict["search"] = getDict(datadict["
                           search"], str(":"))
                else:
                        pass
                return datadict
        except:
                print "An error reading the file occured."
                return 0


def saveFile(filedata, type, mode="w"):
        """saveFile(filedata, type, mode): Takes data & saves it
           into a file in path depending on file type
           & returns '1' or '0'
        """
        import os
        try:
                outfile = open(os.path.join(os.getcwd(), filedata[
                   "project"], type + ".txt"), mode)
                outfile.write("".join(["%s \t%s\n" % (k, v) for k,
                    v in filedata.items()]))
                outfile.close()
                if type == "config":
                        print "File saved. Exiting.... \n"
                        print "Have a nice day."
                        return 1
                else:
                        print "File saved."
                return None
        except:
            print "Saving failed."
            return clearScreen()

def checkTxt(filename, delimiter="\t"):
        """checkTxt(filename, delimiter="\t"): Checks if a file is
             a txt file with 'delimiter' separated fields
           & returns '1' or '0'
        """
        try:
            infile = open(filename, "rb")
            data = infile.read()
            infile.close()
            list = data.splitlines()
            datadict = getDict(list, delimiter)
            if len(datadict) != 0:
                return 1
            else:
                return 0
        except:
            return 0



def filterHTML(URL):
        pass
        """ filterHTML(URL): Gets URL, calls site, parses html,
           searches in tags <description> and <keywords>
                & returns dictionary

        results = {}
        for topic in SearchTopics:
                hits = []
                count = 0
                searchname = "%sSearchTerms" % topic
                #print searchname
                searchterms = globals()[searchname]
                #print searchterms
                for words in searchterms:
                        hits.extend(re.findall(words, feed.lower()
                           ))
                for elem in hits:
                        count += 1
                result[topic] = hits

        return dictResult
        """

def searchSite(email, searchterms, tags= None):
        """searchSite(email, searchterms, tags): Gets a string URL
           , searchterms and, optionally, tags to fetch the
           content of a URL and count the hits on the site for
           each key in searchterms according to respective values
        & returns a dictionary
        """
        import urllib2
        while (len(email) > 0):

                address = 'http://' + email
                request = urllib2.Request(address)
                request.add_header('User-Agent', '
                   OSSResearchProject + http://www.cerna.ensmp.fr/
                   CVs/Eilhard.html')
                print address
                try:
                        opener = urllib2.build_opener()
                        feed = opener.open(request).read()
                except:
                        dump = email.split(".")
                        email = ".".join([elem for elem in dump
                           [1:]])
                        print email
                        print "An error occurred in searchSite."
        print 1

if __name__ == "__main__":
        print info.__doc__
```

Appendix B

Survey Questions

Listing B.1: Survey Questions

```
Dear Madame, Dear Sir,

We are economists (from Cerna, Mines Paristech, France) involved
    in an empirical research project on the governance of open
   source projects listed on SourceForge.net (see our previous
   works on this topic here).

As leaders of open source projects, your answers to the four
   questions listed below would greatly help us in framing our
   analysis. Thank you very much for taking the time to read
   this email and go through the questions.

Using data from SourceForge, our goal is to estimate the
   probability of fixing a bug or feature request over a period
   of 28 months. Keeping all other factors constant, our
   preliminary results suggest that requests submitted by
```

```
    persons with a corporate background take longer to be fixed.
    Your opinion would greatly help us in explaining this
    surprising finding, and more generally in understanding the
    role of the projects leaders. To structure the responses, we
    have formulated four questions:

Question 1:

Do you think that requests, in general, take long because

               ... they are more difficult to solve?

                              Yes/No/No opinion

               ... it is more difficult to find interested
                   developers?

                              Yes/No/No opinion

               ... there is a disagreement between project leaders
                   about that task?

                              Yes/No/No opinion

Question 2:

Do you think that requests by business companies take long to be
```

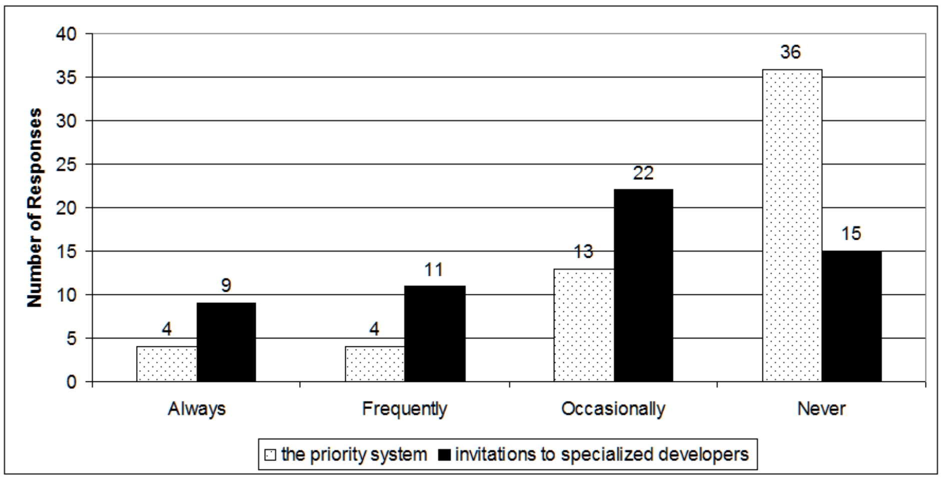
```
    completed?

                    Yes/No/No opinion

If Yes, is it because

          ... they are usually more difficult to solve
              technically than other requests?

                    Yes/No/No opinion

... they are usually less important than other requests in the
    view of developers with other backgrounds?

                    Yes/No/No opinion

          ... there is a disagreement between project leaders
              about that task?

                    Yes/No/No opinion

Question 3:

As a project leader, do you allocate tasks in projects?

... by using the priority system provided by SourceForge, and
    letting developers volunteer
```

```
                              Always / Frequently / Occasionally /
                                 Never

... by inviting specialized developers to work on tasks

                              Always / Frequently / Occasionally /
                                 Never

Question 4:

As a project leader, do you accelerate the fixing of a
   particular task

... by using the priority system provided by SourceForge?

                              Always / Frequently / Occasionally /
                                 Never

... by inviting a specialized developer to work on the task?

                              Always / Frequently / Occasionally /
                                 Never

... by coding yourself?

                              Always / Frequently / Occasionally /
```

```
                                    Never




Again, thank you for your patience and contribution. Of course,
   we would welcome any further comment you may want to make on
   our project and research questions.

If you would like to get updates on our research, please say so
   in your reply.

Yours sincerely,

Jan Eilhard

—

PhD Candidate

Cerna (Center of Industrial Economics)

Mines ParisTech

60 bd Saint Michel, 75272 Paris cedex 6

Tel. : 33 (1) 40 51 92 27

Fax : 33 (1) 44 07 10 46
```

http://www.cerna.ensmp.fr

Figure B.1: How do you distribute requests?



Appendix C

Cross-section Regressions for Corporate Participation Rates

Table C.1: Cross-section Results for Corporate Participation Rate

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	(11)	(12)	(13)	(14)
dur_FRsdmth	0.0269*	0.0271*	0.0270*	0.0270*	0.0268*	0.0262*	0.0263*	0.0265*	0.0265*	0.0262*	0.0262*	0.0264*	0.0264*	0.0268*
	(0.0105)	(0.0105)	(0.0106)	(0.0105)	(0.0106)	(0.0105)	(0.0105)	(0.0105)	(0.0105)	(0.0105)	(0.0105)	(0.0105)	(0.0105)	(0.0105)
dur_FRmeanmth	0.0332+	0.0334+	0.0334+	0.0334+	0.0332+	0.0330+	0.0334+	0.0336+	0.0335+	0.0336+	0.0337+	0.0341+	0.0348+	0.0350+
	(0.0116)	(0.0116)	(0.0116)	(0.0116)	(0.0116)	(0.0115)	(0.0116)	(0.0115)	(0.0115)	(0.0116)	(0.0115)	(0.0116)	(0.0114)	(0.0114)
dur_SRsdmth	-0.0080	-0.0083	-0.0084	-0.0083	-0.0084	-0.0078	-0.0080	-0.0081	-0.0080	-0.0081	-0.0083	-0.0085	-0.0085	-0.0085
	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0109)	(0.0108)	(0.0109)	(0.0108)	(0.0109)
dur_SRmeanmth	-0.0174	-0.0175	-0.0176	-0.0176	-0.0179	-0.0178	-0.0184	-0.0188	-0.0186	-0.0188	-0.0188	-0.0189	-0.0196	-0.0196
	(0.0113)	(0.0113)	(0.0113)	(0.0113)	(0.0112)	(0.0112)	(0.0112)	(0.0112)	(0.0111)	(0.0112)	(0.0112)	(0.0112)	(0.0110)	(0.0110)
dur_BRsdmth	0.0041	0.0040	0.0041	0.0042	0.0046	0.0048	0.0049	0.0050	0.0048	0.0050	0.0050	0.0049	0.0052	0.0047
	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)
dur_BRmeanmth	-0.0032	-0.0033	-0.0034	-0.0034	-0.0027	-0.0027	-0.0024	-0.0024	-0.0025	-0.0024	-0.0025	-0.0027	-0.0024	-0.0029
	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)
r_FRfixed	-0.0455*	-0.0451*	-0.0450*	-0.0450*	-0.0445*	-0.0424*	-0.0426*	-0.0424*	-0.0425*	-0.0424*	-0.0419*	-0.0418*	-0.0406*	-0.0406*
	(0.0183)	(0.0182)	(0.0182)	(0.0182)	(0.0177)	(0.0173)	(0.0174)	(0.0173)	(0.0174)	(0.0174)	(0.0173)	(0.0174)	(0.0171)	(0.0171)
r_SRfixed	0.0065	0.0064	0.0063	0.0063	0.0064	0.0060	0.0060	0.0062	0.0062	0.0061	0.0060	0.0060	0.0058	0.0058
	(0.0069)	(0.0069)	(0.0069)	(0.0069)	(0.0068)	(0.0068)	(0.0068)	(0.0067)	(0.0067)	(0.0067)	(0.0067)	(0.0067)	(0.0067)	(0.0067)
r_BRfixed	0.0081*	0.0081*	0.0082*	0.0082*	0.0088*	0.0086*	0.0087*	0.0087*	0.0087*	0.0087*	0.0090+	0.0090*	0.0090+	0.0090+
	(0.0032)	(0.0032)	(0.0032)	(0.0032)	(0.0035)	(0.0034)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)
licAca	0.4490+	0.4462+	0.4533+	0.4466+	0.4495+	0.4483+	0.4494+	0.4370+	0.4366+	0.4605+	0.4094+	0.3915+	0.3938+	0.3612+
	(0.1187)	(0.1184)	(0.1183)	(0.1179)	(0.1176)	(0.1175)	(0.1175)	(0.1174)	(0.1174)	(0.1168)	(0.1180)	(0.1185)	(0.1173)	(0.1166)
licLGPL	0.1049	0.0901	0.1007	0.1032	0.1065	0.1235	0.1212	0.1295	0.1378	0.1226	0.1103	0.0790	0.1189	0.0976
	(0.1510)	(0.1510)	(0.1505)	(0.1495)	(0.1493)	(0.1480)	(0.1480)	(0.1480)	(0.1473)	(0.1476)	(0.1467)	(0.1481)	(0.1477)	(0.1485)
licOth	0.1046	0.1046	0.1061	0.1046	0.1006	0.0992	0.0379	0.0360	0.0353	0.0448	0.0095	-0.0965	-0.0276	-0.0196
	(0.4167)	(0.4164)	(0.4162)	(0.4163)	(0.4157)	(0.4160)	(0.4231)	(0.4232)	(0.4230)	(0.4446)	(0.4480)	(0.4537)	(0.4539)	(0.4273)
constant	-2.3599+	-2.3588+	-2.3560+	-2.3560+	-2.3644+	-2.3643+	-2.3659+	-2.3606+	-2.3585+	-2.3586+	-2.3480+	-2.3441+	-2.3647+	-2.3507+
	(0.1392)	(0.1392)	(0.1389)	(0.1389)	(0.1393)	(0.1386)	(0.1387)	(0.1384)	(0.1383)	(0.1381)	(0.1385)	(0.1390)	(0.1392)	(0.1383)
N	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643

	(15)	(16)	(17)	(18)	(19)	(20)	(21)	(22)	(23)	(24)	(25)	(26)	(27)	(28)
dur_FRsdmth	0.0270*	0.0267*	0.0265*	0.0264*	0.0262*	0.0261*	0.0261*	0.0260*	0.0262*	0.0258*	0.0257*	0.0256*	0.0254*	0.0255*
	(0.0105)	(0.0105)	(0.0105)	(0.0104)	(0.0105)	(0.0105)	(0.0104)	(0.0105)	(0.0104)	(0.0104)	(0.0104)	(0.0104)	(0.0104)	(0.0104)
dur_FRmeanmth	0.0355+	0.0353+	0.0353+	0.0351+	0.0348+	0.0347+	0.0350+	0.0348+	0.0351+	0.0343+	0.0341+	0.0337+	0.0336+	0.0337+
	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0114)	(0.0113)	(0.0113)	(0.0113)	(0.0113)
dur_SRsdmth	-0.0084	-0.0082	-0.0084	-0.0082	-0.0083	-0.0081	-0.0081	-0.0081	-0.0083	-0.0078	-0.0076	-0.0073	-0.0073	-0.0071
	(0.0109)	(0.0109)	(0.0109)	(0.0108)	(0.0109)	(0.0109)	(0.0108)	(0.0109)	(0.0109)	(0.0108)	(0.0108)	(0.0108)	(0.0108)	(0.0108)
dur_SRmeanmth	-0.0193	-0.0191	-0.0191	-0.0190	-0.0191	-0.0188	-0.0190	-0.0189	-0.0190	-0.0184	-0.0182	-0.0181	-0.0181	-0.0182
	(0.0111)	(0.0111)	(0.0111)	(0.0110)	(0.0111)	(0.0111)	(0.0110)	(0.0111)	(0.0110)	(0.0110)	(0.0110)	(0.0110)	(0.0110)	(0.0110)
dur_BRsdmth	0.0041	0.0041	0.0042	0.0042	0.0044	0.0043	0.0044	0.0045	0.0042	0.0042	0.0043	0.0045	0.0046	0.0046
	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)	(0.0084)
dur_BRmeanmth	-0.0036	-0.0036	-0.0037	-0.0037	-0.0033	-0.0034	-0.0035	-0.0033	-0.0036	-0.0035	-0.0034	-0.0032	-0.0032	-0.0031
	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)	(0.0063)
r_FRfixed	-0.0404*	-0.0402*	-0.0399*	-0.0397*	-0.0408*	-0.0408*	-0.0406*	-0.0410*	-0.0409*	-0.0412*	-0.0416*	-0.0419*	-0.0420*	-0.0412*
	(0.0171)	(0.0170)	(0.0169)	(0.0168)	(0.0172)	(0.0172)	(0.0171)	(0.0172)	(0.0172)	(0.0173)	(0.0172)	(0.0172)	(0.0172)	(0.0171)
r_SRfixed	0.0060	0.0060	0.0060	0.0060	0.0061	0.0062	0.0062	0.0062	0.0063	0.0063	0.0063	0.0064	0.0064	0.0063
	(0.0067)	(0.0067)	(0.0067)	(0.0067)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)	(0.0068)
r_BRfixed	0.0088*	0.0087*	0.0086*	0.0086*	0.0093+	0.0094+	0.0093+	0.0095+	0.0094+	0.0094+	0.0094+	0.0094+	0.0095+	0.0094+
	(0.0035)	(0.0034)	(0.0034)	(0.0034)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)	(0.0035)
licAca	0.3348+	0.3149+	0.3212+	0.3013+	0.2860*	0.2671*	0.2686*	0.2720*	0.2544*	0.2439*	0.2449*	0.2373*	0.2541*	0.2732*
	(0.1162)	(0.1158)	(0.1149)	(0.1150)	(0.1147)	(0.1146)	(0.1141)	(0.1131)	(0.1122)	(0.1119)	(0.1113)	(0.1110)	(0.1102)	(0.1096)
licLGPL	0.0517	0.1237	0.1705	0.1997	0.1886	0.2209	0.2381	0.2478	0.2212	0.2108	0.2379	0.2799	0.3507*	0.3918*
	(0.1530)	(0.1519)	(0.1505)	(0.1500)	(0.1510)	(0.1506)	(0.1505)	(0.1521)	(0.1551)	(0.1557)	(0.1559)	(0.1554)	(0.1528)	(0.1529)
licOth	-0.0900	-0.0308	0.2023	0.1633	0.1519	0.0633	0.0657	-0.0927	-0.0975	-0.1318	-0.3609	-0.3978	-0.4199	-0.4125
	(0.4305)	(0.4302)	(0.4072)	(0.4052)	(0.4051)	(0.4033)	(0.4034)	(0.4399)	(0.4397)	(0.4378)	(0.4645)	(0.4640)	(0.4634)	(0.4635)
constant	-2.3429+	-2.3515+	-2.3496+	-2.3481+	-2.3423+	-2.3454+	-2.3495+	-2.3519+	-2.3384+	-2.3347+	-2.3374+	-2.3396+	-2.3507+	-2.3597+
	(0.1372)	(0.1374)	(0.1376)	(0.1377)	(0.1376)	(0.1373)	(0.1377)	(0.1383)	(0.1382)	(0.1380)	(0.1381)	(0.1382)	(0.1382)	(0.1379)
N	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643	2,643

Standard errors in parentheses * p<.05 + p<.01

Bibliography

Amabile, T. (1983). The social psychology of creativity: A componential conceptualization. *Journal of Personality and Social Psychology*, 45(2):357–376.

Baldwin, C. and Clark, K. (2006). Does code architecture mitigate free riding in the open source development model. *Management Science*, 52(7).

Behlendorf, B. (1999). Open source as a business strategy. In DiBona, C., Ockman, S., and Stone, M., editors, *Open Sources: Voices from the Open Source Revolution*, chapter 11, pages 149–170. O'Reilly & Associates.

Belenzon, S. and Schankerman, M. A. (2008). Motivation and sorting in open source software innovation. *SSRN eLibrary*.

Benkler, Y. (2002). Coase's penguin, or, linux and the nature of the firm. *Yale Law Journal*, 112(3):367–445.

Bessen, J. E. (2005). Open source software: Free provision of complex public goods. *SSRN eLibrary*.

Bessen, J. E. and Maskin, E. S. (2000). Sequential innovation, patents, and imitation. *SSRN eLibrary*.

Bitzer, J. (2004). Commercial versus open source software: the role of product heterogeneity in competition. *Economic Systems*, 28(4):369–381.

Boehm, B. (1981). Software engineering economics. *Englewood Cliffs*.

Bonaccorsi, A. and Rossi, C. (2003a). Contributing to the common pool resources in open source software. a comparison between individuals and firms.

Bonaccorsi, A. and Rossi, C. (2003b). Licensing schemes in the production and distribution of open source software: An empirical investigation. *SSRN eLibrary*.

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

Brooks, F. (1978). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA.

Casadesus-Masanell, R. and Ghemawat, P. (2006). Dynamic mixed duopoly: A model motivated by linux vs. windows. *Management Science*, 52(7):1072 – 1084.

Chesbrough, H. (2003). *Open Innovation: The New Imperative for Creating and Profiting from Technology*. Harvard Business School Press.

Christensen, L. and Greene, W. (1976). Economies of scales in us electronic power generation. *Journal of Political Economy*, 84:655–676.

Coase, R. (1974). The lighthouse in economics. *Journal of Law and Economics*, 17(2):357–376.

Cohen, W., Nelson, R., and Walsh, J. (2000). Protecting their intellectual assets: Appropriability conditions and why US manufacturing firms patent (or not). *NBER Working Paper*.

Dahlander, L. (2005). Appropriation and appropriability in open source software. *International Journal of Innovation Management*, 9(3):259–285.

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

Dahlander, L. and Wallin, M. (2006). A man on the inside: Unlocking communities as complementary assets. *Research Policy*, 35:1243–1259.

David, P. A. and Rullani, F. (2006). Micro-dynamics of free and open source software development: Lurking, laboring and launching new projects on sourceforge. *Stanford Institute for Economic Policy Research Mimeo*.

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.

Eilhard, J. (2008a). Loose contracts, tight control? Firms on SourceForge. *Cerna Working Paper*.

Eilhard, J. (2008b). Open source incorporated. *Cerna Working Paper*.

Evans, D. and Layne-Farrar, A. (2004). Software patents and open source: The battle over intellectual property rights. *SSRN eLibrary*.

Fershtman, C. and Gandal, N. (2008). Microstructure of Collaboration: The Network of Open Source Software. *SSRN eLibrary*.

Fosturi, A., Giarratana, M., and Luzzi, A. (2008). The penguin has entered the building: The commercialization of open source software products. *Organization Science*, 19(2):292–328.

Gao, Y., Van Antwerp, M., Christley, S., and Madey, G. (2007). A research collaboratory for open source software research. Available at: http://www.nd.edu/ oss/Papers/FLOSS07.pdf.

Gartner (2008). Gartner says as number of business processes using open-source software increases, companies must adopt and enforce an oss policy. Available at: http://www.gartner.com/it/page.jsp?id=801412.

Ghosh, R., Glott, R., Krieger, B., and Robles, B. (2002). Free/libre and open source software: Survey and study (floss), final report, Part 4: Survey of developers. Technical report. available at: http://www.infonomics.nl/FLOSS/report.

Giuri, P., Ploner, M., Rullani, F., and Torrisi, S. (2010). Skills, division of labor and performance in collective inventions: Evidence from open source software. *International Journal of Industrial Organization*, 28(1):54 – 68.

Griliches, Z. and Mairesse, J. (1998). Production functions: The search for identification. pages 169–203.

Grossman, S. and Hart, O. (1986). The costs and benefits of ownership: A theory of vertical and lateral integration. *The Journal of Political Economy*, 94(4):691–719.

Haaland, K., Sowe, S. K., Ghosh, R., and David, P. A. (2009). Investigating the productivity of open source software developers. Available at: http://www.decon.unipd.it/personale/curri/manenti/floss/haaland.pdf.

Hammond, J., Gerush, M., and Silekis, J. (2009). Open source software goes mainstream. *Forrester Enterprise and SMB Software Survey*.

Hausman, J., Hall, B., and Griliches, Z. (1984). Econometric models for count data with an application to the patents-R&D relationship. *Econometrica*, 52(4):909–938.

Hawkins, R. (2004). The economics of open source software for a competitive firm. *NETNOMICS*, 6(2):103–117.

Hecker, F. (1999). Setting up shop: The business of open-source software. *Software, IEEE*, 16(1):45–51.

Heide, J. and John, G. (1988). The role of dependence balancing in safeguarding transaction-specific assets in conventional channels. *The Journal of Marketing*, pages 20–35.

Henkel, J. (2006). Selective revealing in open innovation processes: The case of embedded linux. *Research Policy*, 35(7):953–969.

Henkel, J. (2008). Champions of revealing - the role of open source developers in commercial firms. *SSRN eLibrary*.

Hennart, J. (1988). A transaction costs theory of equity joint ventures. *Strategic Management Journal*, pages 361–374.

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

Jaaksi, A. (2006). Building consumer products with open source. Available at: http://www.linuxdevices.com/articles/AT7621761066.html.

Jaaksi, A. (2009). How do we build maemo devices? Available at: http://jaaksi.blogspot.com.

Jeon, D.-S. and Petriconi, S. (2009). Dynamics of knowledge-based service industry and open source.

Jeppesen, L. and Frederiksen, L. (2006). Why do users contribute to firm-hosted user communities? *Organization Science*, 17(1):45–63.

Joskow, P. (1985). Vertical integration and long-term contracts: The case of coal-burning electric generating plants. *Journal of Law, Economics, and Organization*, 1(1):33–80.

Klein, B., Crawford, R., and Alchian, A. (1978). Vertical integration, appropriable rents, and the competitive contracting process. *Journal of Law and Economics*, pages 297–326.

Kroah-Hartman, G., Corbet, J., and McPherson, A. (2009). Linux kernel development: How fast it is going, who is doing it, what they are doing, and who is sponsoring it (an august 2009 update). *Linux Foundation*.

Lakhani, K. and Wolf, R. (2005). Why hackers do what they do: Understanding motivation and effort in free/open source software projects. In Feller, J., Fitzgerald, B., Hissam, S. A., and Lakhani, K. R., editors, *Perspectives on Free and Open Source Software*, chapter 1, pages 3–22. MIT Press, Cambridge.

Lamastra, C. (2009). Firms' participation in open source projects: Which impact on software quality and success. *Working Paper*. available at: http://www2.druid.dk/conferences/viewpaper.php?id=6067&cf=32.

Langois, R. and Gazarelli, G. (2008). Of hackers and hairdressers: Modularity and the organizational economics of open-source collaboration. *Industry and Innovation*, 15(2):125–143.

Laurent, A. (2004). *Understanding open source and free software licensing.* O'Reilly Media, Inc.

Lerner, J., Pathak, P., and Tirole, J. (2006). The dynamics of open-source contributors. *The American Economic Review*, pages 114–118.

Lerner, J. and Tirole, J. (2002). Some simple economics of open source. *Journal of Industrial Economics*, 50(2):197–234.

Lerner, J. and Tirole, J. (2005). The economics of technology sharing: Open source and beyond. *Journal of Economic Perspectives*, 19(2):99–120. Retrieved from http://ideas.repec.org/a/aea/jecper/v19y2005i2p99-120.html.

Lévêque, F. and Ménière, Y. (2006). Copyright versus patents: The open source software legal battle. *CERNA Working Paper*.

Marschak, J. and Andrews, W. (1944). Random simultaneous equations and the theory of production. *Econometrica*, 12(3):143–172.

McGovan, D. (2001). Legal implications of open-source software. *University of Illinois Law Review*, 1:241–304.

Muntz, G. (1909). The relation between science and practice and its bearing on the utility of the institute of metals. *Journal of the Institute of Metals*, 1:289–291.

Mustonen, M. (2005). When does a firm support substitute open source programming? *Journal of Economics & Management Strategy*, 14(1):121–139.

Olley, G. and Pakes, A. (1996). The dynamics of productivity in the telecommunications equipment industry. *Econometrica*, 64(6):1263–1297.

Olson, M. (1971). *The Logic of Collective Action.* Harvard Universtity Press, Cambridge, MA.

O'Mahony, S. and West, J. (2005). What makes a project open source? mitigating from organic to synthetic communities. *Academy of Management Annual Meeting, Honnolulu (August).*

Osterloh, M. and Rota, S. (2007). Open source software development-just another case of collective invention? *Research Policy*, 36(2):157–171.

Papke, L. and Wooldridge, J. (1996). Econometric methods for fractional response variables with an application to 401 (k) plan participation rates. *Journal of Applied Econometrics*, pages 619–632.

Perens, B. (1999). The open source definition. In DiBona, C., Ockman, S., and Stone, M., editors, *Open Sources: Voices from the Open Source Revolution*, chapter 12, pages 171–188. O'Reilly & Associates.

Perens, B. (2006). The monster arrives: Software patents lawsuits against open source developers. available at: http://technocrat.net/d/2006/6/30/5032.

Raymond, E. (2001). *The cathedral and the bazaar. Musings on Linux and Open Source by an accidental revolutionary.* O'Reilly.

Rosen, L. (2001). Which open source licens should I use for my software? available at:http:/:www.rosenlaw.com/html/GL5.pdf.

Rosen, L. (2004). *Open Source Licensing: Software Freedom and Intellectual Property Law*. Prentice Hall PTR.

Schmidt, K. and Schnitzer, M. (2003). Public subsidies for open source? some economic policy issues of the software market. *CEPR Discussion Papers*.

Scotchmer, S. and Maurer, S. (2006). Open source software: The new intellectual property paradigm. In Hendershott, T., editor, *Handbook of Economics and Information Systems*, pages 285–319. Elsevier, Amsterdam.

Shah, S. (2000). Sources and patterns of innovation in a consumer products field: Innovations in sporting equipment. *Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, WP-4105*.

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

Shelanski, H. and Klein, P. (1995). Empirical research in transaction cost economics: a review and assessment. *Journal of Law, Economics, and Organization*, 11(2):335–361.

Singh, P. V., Tan, Y., and Mookerjee, V. (2008). Network effects: The influence of structural social capital on open source project success. *SSRN eLibrary*.

SourceForge (2010). available at: http://www.sourceforge.net.

UNU-MERIT (2006). Study on the: Economic impact of open source software on innovation and the competitiveness of the information and communication technologies (ict) sector in the eu. Technical report, UNU-MERIT, the Netherlands.

Valloppillil, V. (1998). Halloween documents. *Internal Strategy Microsoft Memorandum*. avaliable at: http://www.catb.org/ esr/halloween/halloween1.html.

Ven, K. and Mannaert, H. (2007). Challenges and strategies in the use of open source software by independent software vendors. *Information and Software Technology*, pages 991–1002.

Välimäki, M. (2003). Dual licensing in open source software industry. *Systemes d'Information et Management*, 1:2003.

von Hippel, E. (1988). *The sources of innovation.* Oxford University Press, New York.

von Hippel, E. (2001). Innovation by user communities: Learning from open-source software. (cover story). *MIT Sloan Management Review*, 42(4):82–86.

von Hippel, E. and Katz, R. (2002). Shifting innovation to users via toolkits. *Management Science*, 48(7):821–833.

von Hippel, E. and von Krogh, G. (2003). Open source software and the "private-collective" innovation model: Issues for organization science. *Organization Science*, 14(2):209–223.

von Krogh, G., Spaeth, S., and Lakhani, K. R. (2003). Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241.

Weiss, A. (2005). Real world open source: The TCO question. *Serverwatch.* available at: http://www.serverwatch.com/tutorials/article.php/3529871.

West, J. (2003). How open is open enough?: Melding proprietary and open source platform strategies. *Research Policy*, 32(7):1259–1285.

Wheeler, D. (2002). More than a gigabuck: Estimating gnu/linux's size. available from: http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html.

Wichmann, T. (2002a). Free/libre and open source software: Survey and study (floss), final report, part I: Use of open source software in firms and public institutions. Technical report.

Wichmann, T. (2002b). Free/libre and open source software: Survey and study (floss), final report, part II: Firms' open source activities-motivations and policy implications. Technical report.

Williamson, O. (1967). Hierarchical control and optimum firm size. *The Journal of Political Economy*, 75(2):123–138.

Williamson, O. (1991). Comparative economic organization: The analysis of discrete structural alternatives. *Administrative science quarterly*, 36(2):219–44.

Williamson, O. (1996). Economics and organization: A primer. *California Management Review*, 38(2):131–146.