Code Reuse in Open Source Software

Stefan Haefliger
Georg von Krogh
Sebastian Spaeth*
ETH Zurich, Switzerland

Forthcoming in Management Science

Abstract

Code reuse is a form of knowledge reuse in software development, which is fundamental to innovation in many fields. Yet, to date, there has been no systematic investigation of code reuse in open source software projects. This study uses quantitative and qualitative data gathered from a sample of six open source software projects to explore two sets of research questions derived from the literature on software reuse in firms and open source software development. We find that code reuse is extensive across the sample and that open source software developers, much like developers in firms, apply tools that lower their search costs for knowledge and code, assess the quality of software components, and they have incentives to reuse code. Open source software developers reuse code because they want to integrate functionality quickly, because they want to write preferred code, because they operate under limited resources in terms of time and skills, and because they can mitigate development costs through code reuse.

Keywords: Innovation, Private-collective innovation model, Incentives, Software development, Knowledge reuse, Software reuse, Open Source software

*We would like to thank Alessandro Rossi, Eric von Hippel, Wally Hopp, Mark Macus, Christian Loepfe, Karim Lakhani, Margit Osterloh, Ivan von Warthburg, Rishab Ghosh, Marco Zamarian, Giovanna Devetag, Joel West, participants at the ROCK seminar in Trento, the Conference on Lead User Innovation at the Sloan School of Management, MIT, and all participating members of the free software/open source community for their feedback and valuable input. Anonymous reviewers and an Associate Editor for Management Science repeatedly offered most valuable input. This research was supported by the Swiss National Science Foundation (grant 100012-101805). All authors contributed equally. Address of the corresponding author: sspaeth@ethz.ch; Kreuzplatz 5, CH-8032 Zurich, +41 44 632 8837.

1. Introduction

The particular context of open source software development[1], its organization in worldwide informal and virtual communities as it is Internet-based, the mostly public and archived communication between developers, and the availability of the code base have contributed to the general interest of researchers from many fields. The extraordinary success of some of the resulting software products (such as GNU/Linux, Apache, Bind DNS server, OpenOffice, Mailman) has drawn attention from the public and both software-creating and software-using organizations to this way of developing software. Yet, not all open source projects produce software targeted directly at the end-user. Some software is designed to be reused and to provide functionality to other software projects. For example, Lame, a music encoder, cannot be used directly but has to be built into, and used by, another program to create mp3 files. This leads to the question: do open source software developers tend to build software from scratch or do they rather reuse readily available knowledge and software *code* from other projects? On the one hand, free and open source software licenses, such as the GNU General Public License (GPL), grant permission to reuse software components within the limits of the license and one should expect open source software developers to build on each other's work. On the other hand, the many barriers to code reuse discovered in firms (Lynex and Layzell, 1998) raise doubts about the actual reuse behavior of developers in the absence of corporate reuse programs. Motivated by these two contradictory premises, this paper derives two sets of research questions from the literature on code reuse in firms and on open source software development and explore them using qualitative and quantitative data from 6 projects that vary in project agenda, age, size, and other factors.

Based on the premise that software development and code reuse in particular, hinges on technical as well as non-technical issues (Kim and Stohr, 1998), our analytical

[1]For better readability the term open source will be used throughout this article, but the study also refers to Libre and Free software which shares the same technical definition, but is driven by philosophical/moral considerations on freedom rather than technical arguments: see http://www.gnu.org/philosophy/free-sw.html.

starting point is the behavior of individual developers. The general interest of this paper is to explore the practices of knowledge, and, in particular, code reuse in open source software development; what is being reused, when is it reused, by whom, and for what reasons.

The paper is organized as follows. Section 2 briefly discusses relevant theory and research on knowledge and code reuse in innovation and software development and identifies the research gap in the area of open source software development. We formulate research questions from the literature that relate to code reuse in open source software development. Section 3 gives an overview of the research method and the sample of projects studied. Section 4 presents the findings in the form of an inventory of code reuse across the project sample, and we complement these with findings regarding other types of knowledge reuse. This section also explores the research questions developed in Section 2. Finally, Section 5 concludes the paper and discusses implications of the study for management practice and proposes future research topics founded on this work.

2. Research gap: Open source software, knowledge, and code reuse

Open source software development is an example of "private-collective" innovation (von Hippel and von Krogh, 2003): software developers derive private benefits from writing software and sharing their code and collectively contribute to the development of software. Such private benefits include enjoyment, fun, learning, reputation, and community membership (Lakhani and Wolf, 2005; Hertel, et al. 2003). An assertion in the private-collective view of open source software innovation is that benefits gained from contributing to the public good must outweigh the privately incurred cost of contributing to the software development (von Krogh and von Hippel, 2006).

The literature on technological innovation argues that knowledge reuse is an important mitigating factor for the cost of innovation (Langlois, 1999). Returns on investment in the creation of new knowledge hinges on the extent to which this knowledge can be applied across the development of new processes and products. Therefore, one of the central problems in the management of innovation is if and how firms reuse previously created knowledge across the various stages of an innovation process (see Argote, 1999; Majchrzak, Cooper, and Neece, 2004; Zander and Kogut, 1995).

The practice of knowledge reuse has been particularly relevant for innovation in the software industry and it is here that many of the most significant advances in the research on knowledge reuse have been made (Cusumano, 1991; Markus, 2001). While software *code* is notably explicit knowledge that is both readable by humans and enables a computer to perform specific functions, knowledge reuse may cover more than code reuse (Knight and Dunn, 1998: p. 295). As Barnes and Bollinger (1991, p.14) suggest: "The defining characteristic of good reuse is not the reuse of software *per se*, but the reuse of human problem-solving." Several types of knowledge can be reused across the different stages of software development (Frakes and Isoda, 1994): problem description, artifacts, project proposals, feasibility reports, enterprise models, data dictionaries, prototypes, decision tables, pseudo-code, source code, databases, the tacit knowledge of developers, networks of developers, and so on (for an overview, see Cybulski et al, 1998; Prieto-Diaz, 1993; Ravichandran, 1999). The documentation of software design patterns facilitates the reuse of problem solving in software engineering, particularly when using object-oriented languages (Gamma et al., 1995; Schmidt, et al 1996). The reuse of software is enabled through modular software architectures and the development of generic software components. However, the design of generic components requires substantial investment for a firm that can only pay off in the long run if and when the firm saves development costs through component reuse in software projects (Banker and Kauffman, 1991). In the software industry, firms that reuse code on more than one project can amortize development costs faster and reduce development time in new projects (Barnes et al., 1987; Banker and Kaufmann, 1991). Reusing code and components from software libraries also enhances the quality of new software products by allowing for fully tested and debugged software (Knight and Dunn, 1998).

In spite of the reported benefits, several studies on software development firms have found that code reuse in software development is problematic and that the success of corporate reuse programs hinges on organizational factors more than on technical factors (Apte et al., 1990; Isoda, 1995; Kim and Stohr, 1998; Rothenberger et al., 2003). This literature also provides insights regarding the possibilities of code reuse in open source software: in software development firms, corporate reuse programs need to commit an initial investment to reuse (Isoda, 1995) in order to generate long-term savings including life-cycle benefits such as maintenance (Banker et al., 1993; Basili, 1990). Program success depends on standards and tools provided to developers (Lim, 1994; Kim and Stohr, 1998), on the certification of software (Knight and Dunn, 1998), as well as on the incentives for developers to reuse (Poulin, 1995).

Systematic reuse in software development firms requires years of investment (Frakes and Isoda, 1994) in order to create and maintain reusable code and other knowledge (Lim, 1994), populate repositories and libraries (Griss, 1993; Poulin, 1995), and provide tools for developers to identify and reuse code (Isakowitz and Kauffman, 1996). The organization's funding structure usually needs adaptation to coordinate reuse investments across the organization (Lynex and Layzell, 1998), because developers need to work in repositories and components that are not directly linked to a product. Pilot programs, accompanied by code reuse performance metrics (Frakes and Terry, 1996), may instigate systematic reuse that can be monitored across the organization (Banker et al., 1993). Management needs to appoint champions as sponsors, reuse-librarians or -coordinators (Isakowitz and Kauffman, 1996; Joos, 1994; Kim and Stohr, 1998).

The success of a corporate reuse program depends on whether the costs to the developer of search and integration are lower than the costs of writing the software from scratch (Banker et al., 1993). According to the literature, this can be achieved by creating standards and tools that facilitate the search for and integration of software components. Elaborate classification schemes (Isakowitz and Kauffman, 1996) facilitate the use of and access to libraries and lower search costs for developers. Domain analyses, documentation, and quality standards enhance the ability to reuse software components (Poulin, 1995). Ideally, the information accompanying reusable code should incorporate a quality rating or certification in order to enhance the developer's trust in code and components written by someone else (Knight and Dunn, 1998; Poulin, 1995). This emphasis on quality stems from the software developer "(feeling) that defects in a reused code could have a substantial negative impact on whatever system he or she is building" (Knight and Dunn, 1998: 293).

Incentives play a crucial role in corporate reuse programs (Isoda, 1995; Lynex and Layzell, 1998; Poulin, 1995; Tracz, 1995). The monetary or reputation-based incentives offered by software development firms (Poulin, 1995) need to outweigh the dominant notion that code reuse is "boring" or "less satisfying" than writing code (McClure, 2001; Tracz, 1995), overcome the not-invented-here syndrome, and the general resistance to change in organizations (Lynex and Layzell, 1998).

In contrast to software development firms, open source software development projects do not feature corporate reuse programs and usually have no financial resources to invest in tools, standards, and incentives. This could have adverse effects on code reuse; for instance, the lack of incentives could prevent systematic code reuse by open source software developers who are known to code for the creative challenge and the fun of tackling "technically sweet" problems (see Raymond, 2000: 25; Lakhani and Wolf, 2005; Hertel et al., 2003). Thus, in the absence of corporate reuse programs, it can be reasoned that open source software development projects would need “equivalent” mechanisms to substitute such programs. Based on the review of the literature on code reuse in software development firms, the following questions can be formulated regarding such mechanisms:

Research question 1 a: Do equivalents to standards and tools (found in software development firms) support code reuse in open source software development?

Research question 1 b: Do equivalents to quality ratings and certificates (found in software development firms) support code reuse in open source software development?

Research question 1 c: Do equivalents to incentives (found in software developing firms) support code reuse in open source software development?

Despite the absence of corporate reuse programs, research on open source software development provides reasons to expect systematic code reuse among developers. Three leading reasons are examined in this study. First, the demanding requirements for the functionality and architecture of the code after the inception of an open source project might make it rational for developers to reuse already existing code. Second, the desire to work on preferred tasks should lead developers to reuse code that they prefer not to write on their own. Third, resource constraints in terms of time and skills should lead to reuse behavior in open source software development.

The existence of a code base seems crucial in order to mobilize open source developers, as shown for example in a study of the Freenet project (von Krogh et al., 2003). After a project's inception, its developers have to fulfill what we call a "*credible promise,*" best defined by using a quote from Lerner and Tirole (2002: 220): "a critical mass of code to which the programming community can react. Enough work must be done to show that the project is doable and has merit." The credible promise enables sufficient functionality of the software to catch the interest of potential users and developers. A side effect of reusing components is its impact on the software architecture which is also evaluated by prospective developers (Baldwin and Clark, 2006). The reuse of a software component takes advantage of a design option (Baldwin and Clark, 2006; Favaro et al., 1998, MacCormack et al., 2006) and adds to the modularity of the overall architecture. Given the advantages of modularity in software development across the time span of the project (Baldwin and Clark, 2000; Garud and Kumaraswamy, 1995), the reuse of components seems rational at any time, not only during the early phase of a project.

Developers of open source software are known to self-assign to tasks based on their preferences and ability (Benkler, 2002; Bonaccorsi and Rossi, 2003; Yamauchi, et al., 2000) and they seek a creative challenge and fun when writing software (Lakhani and Wolf, 2005; Hertel et al., 2003). Yet, some essential tasks in open source software development are considered to be mundane and boring (Shah, 2006; von Hippel and Lakhani, 2003). A solution to achieving an operational software product could be code reuse. Developers who face several essential tasks may choose to solve the less preferred ones by reusing code rather than writing everything from scratch.

Any open source software developer can consider the vast amount of available open source software when building their own code base. Open source licenses convey the basic rights to the developer to retrieve the code, inspect and modify it, and to freely redistribute modified or unmodified versions of the software to others. Such a license inherently encourages a developer to reuse code, although a license might require that derivative works are released under an open source license as well[2]. The cost of contributing to open source software development can be substantial. For example, those who want to join a community of developers must demonstrate considerable skill at solving technical problems. Von Krogh, Spaeth, and Lakhani (2003) found that newcomers to a project reused software they had written for other projects in order to make their first contributions. Contributions to the public good incur private costs to developers (von Hippel and von Krogh, 2003). Hence, one should expect that developers find it opportune to mitigate their development costs through code reuse from other projects. Empirical studies of Apache and Linux developers have demonstrated that a strong incentive for developing open source software is to solve a technical problem by writing software code and getting feedback from other users (von Hippel, 2001; Hertel et al. 2003). If software that solves the problem is already available under an open source license, there is no reason why a developer should write their own

[2]The exact definition of what poses a derivative work is disputed. There are, for example, many ways in which programs can interact. What kind of interaction implies a derivative work, versus a mere aggregation of individual programs, is a controversial issue among the various factions of producers and consumers of open source software.

code. This economic logic should apply beyond the initial release of the software. Hence, the following questions can be formulated:

Research question 2 a: Does the open source software developers' aim to publish workable software as early as possible (credible promise) supports code reuse in open source software development?

Research question 2 b: Does the open source software developers' self-assignment to tasks according to their preferences and ability supports code reuse in open source software development?

Research question 2 c: Do Open source software developers' resource constraints (in terms of limited time and skills) supports code reuse in open source software development?

In sum, knowledge and code reuse are fundamental to the economics of innovation and central to software development. The characteristics of open source software development could provide both favorable and inauspicious conditions for code reuse but, to date, there is no empirical research available on the topic.

3. Research Method and Sample

This section describes the sample selection process and the research method that guided this study. The literature review on code reuse and open source software led to the formulation of precise research questions for code reuse. The available literature indicated reasons why code reuse might occur in open source software development without providing empirical evidence. Thus, we proceeded to explore the questions using a multiple case study design drawing upon several different data sources (Yin, 1989). An open invitation to a short, anonymous, web-based survey was posted on a developer mailing list in order to decide whether the topic was of any interest and relevance to the field. The resulting 30 replies indicated that knowledge and code reuse are important issues and an integral part of open source software development practice. Next, interviews were conducted and emails, public documents, and code were gathered from an initial sample of 15 projects. The sample included a wide variety of software products such as office software, games, a hardware driver, and an instant messenger client. The projects needed to fulfill three conditions to be included in the sample: 1) the project was under active development, allowing us to track its development activity and interview key developers, 2) the source code modifications of the project needed to be available online, and 3) the project had to have been in existence for at least a year which enabled us to track code reuse over time.

For a more in-depth analysis, the initial sample was reduced to a "core sample" of 6 projects exhibiting variance on the sampling criteria: size (lines of code (LOC), number of developers), objective, date of inception, target audience for software product, license, and programming language. By keeping the high variety of project characteristics, a sampling bias was avoided (e.g., Stake, 1995). Moreover, in order for projects to be included in the core sample, their core developers needed to be available for interviews. The resulting core sample is presented in Table 1.

The data sources from the core sample included interviews, source code, code modification comments, mailing lists, and various web pages related to the projects. Between December 2003 and June 2004, interviews were conducted with 21 core developers[3] of sample projects, 12 of which belonged to the core sample[4]. The interviews were semi-structured, conducted by telephone, and lasted on average 50 minutes. The developers were contacted by email first. Two thirds of the interviews requested were carried out. The interviewees of the core sample consisted of members of the inner circle of the current development team. We were able to interview the developers of 74% of all instances of initial software component code imports into the core sample projects (referred to as "architectural reuse," see Section 4). In order to protect their privacy, the names of the respondents are replaced by capital letters throughout this text. Developer interviews were used to increase familiarity with the project, clarify open issues, and to examine the motivation for knowledge and code reuse.

---

[3]A "core-developer" has CVS access, contributes the bulk of the code, and assumes administrative tasks. See also von Krogh, Spaeth, and Lakhani (2003) and Shah (2006).
[4]For 4 of the six projects, 2 developers were interviewed. For GNUnet only the founder (who wrote the bulk of the project's code) was interviewed and for xfce4, 3 developers were interviewed.

Project	Objective	Lines of code	Inception	Developers *	Target Audiences	Licenses **	Programming Languages
Xfce4	*Xfce is a lightweight desktop environment for unix-like operating systems. It aims to be fast and lightweight, while still being visually appealing and easy to use. Xfce 4 is a complete rewrite of the previous version. It's based on the GTK+ toolkit version 2.*	2,435,172	2001	17	End-user	BSD, GPL	C
TikiWiki	*Tiki CMS/Groupware (aka TikiWiki) is a powerful open-source Content Management System (CMS) and Groupware that can be used to create all sorts of Web Applications, Sites, Portals, Intranets and Extranets. TikiWiki also works great as a web-based collaboration tool. It is designed to be international, clean, and extensible.*	842,025	2002	89	End-user / developer	LGPL	Javascript, PHP
Abiword	*AbiWord is a free word processing program similar to Microsoft® Word. It is suitable for typing papers, letters, reports, memos, and so forth.*	1,368,264	1998	63	End-user	GPL	C, C++, Objective C
GNUnet	*GNUnet is a framework for secure peer-to-peer networking that does not use any centralized or otherwise trusted services. A first service implemented on top of the networking layer allows anonymous censorship-resistant file-sharing. GNUnet uses a simple, excess-based economic model to allocate resources. Peers in GNUnet monitor each others behavior with respect to resource usage; peers that contribute to the network are rewarded with better service.*	259,586	2001	16	End-user / developer	GPL	C
Irate	*iRATE radio is a collaborative filtering system for music. You rate the tracks it downloads and the server uses your ratings and other people's to guess what you'll like. The tracks are downloaded from websites which allow free and legal downloads of their music.*	109,201	2003	21	End-user	GPL	Java
OpenSSL	*The OpenSSL Project is a collaborative effort to develop a robust, commercial-grade, full-featured, and Open Source toolkit implementing the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols as well as a full-strength general purpose cryptography library.*	1,014,816	1999	12	Developer	Apache-style	C

* Number of active or core developers as stated by the project ** Details on licenses see e.g. http://opensource.org/licenses

Table 1: Core sample overview

The developer mailing lists of all core sample projects were analyzed two weeks prior to and after the first reuse of a component. We coded all reuse-related comments and measured the length of discussions (by anyone on the lists) spurred by the reuse incidents.

The source code of the core sample was initially available on project websites and managed using the Concurrent Versions System (CVS) source code management tool (for a description, see von Krogh et al. 2003). The CVS source code repositories of the core sample were retrieved and stored in a local database in order to enable the analysis of source code changes and the associated comments in the CVS[5]. The source code was examined in four ways: accredited lines of code reuse, identification of reused components, identification

[5] All projects were recorded from their inception until mid-2004. An exception was Xfce4 which is a rewrite of Xfce3. Here, the inception date was adjusted to the time when development actually picked up and the developers migrated to working on Xfce4. This meant relocating 268 out of 62,000 CVS incidents to the new inception date.

of functions within the components and their reuse across the core sample, and an authorship analysis.

First, a rough analysis was performed, including how many lines of software code were directly 'copied and pasted' from other projects. In order to find these, the originating project and/or authors needed to be accredited in the CVS comment and, therefore, they were referred to as “accredited lines of code-” reuse. Developers commented and accredited these lines, such as the following comment made by developer G: *"added configuration file parsing without OpenSSL using code from xawtv."* Based on this analysis[6], 38,245 accredited lines of code were identified across the core sample, - a relatively low value compared to more than 6 million lines of code in the core sample projects. The developers commented that copying lines of code only occurred infrequently and in small quantities, but that giving credit was mandatory. However, there might still be an unknown quantity of imported lines of code which was not explicitly accredited.

The identification of reuse of software components was done in an automated manner by filtering the source code for programming statements used to include components[7]. In the next step, the functions within each reused component were identified and a more fine-grained analysis identified the reuse of functions offered by the components identified across the core sample. The tool “*Doxygen*[8]” was used to extract the software architecture, specifically the *application programming interface* (API) of the identified components. This XML file-based information was used to search all code modifications of the six sample projects for function calls added to each project's software, using functionality

[6]In order to allow for the identification of accredited lines of code (ALOC), we applied a Bayesian filter. This is based on an algorithm which estimates the probability of a code modification to be a knowledge reuse incident, using conditional probabilities, by rating the occurrence of specific words based on training data. In this study, the filter was trained by manually analyzing the source code modifications of two projects. We were, thus, able to calculate the probability that a source code modification comment related to imported lines of code from another project. Resulting hits and probable hits were examined manually in order to settle whether or not they represented ALOC reuse. As a reviewer correctly pointed out, this method provides the lower bounds for ALOC reuse as the filter might miss actual ALOCs. For the Bayesian filter reference go to: http://www.paulgraham.com/spam.html

[7]The search included source/header files through statements such as '*#include*' for C/C++, '*package*'/'*import*' for Java, and '*require*'/'*require_once*'/'*include*'/'*include_once*' for php-based projects. System files, such as files belonging to the standard C library or the Linux kernel, were filtered out.

[8] Doxygen is described as “a documentation system for C++, C, Java, Objective-C, IDL (Corba and Microsoft flavors) and to some extent PHP, C#, and D.” It is publicly available at http://doxygen.org.

provided by the included components. This result covered high-level calls using the public API of the components.

Using the first occurrence of each included file or reused function, the analysis identified 2,975 unique reuse incidents. Of these incidents, 200 imported a component (or part of a component) and 2,775 incidents made use of the functions offered by the reused components. A total of 55 reused components were identified in this way, leading to a component reuse inventory (as shown in Table 2). The identified list of software components was sent back to the project lead developers (listed on the projects' web pages) in order to check its validity. Out of 6 projects, 4 replies were received validating the results. One respondent confirmed in the interview that the project only reuses one optional component. In one case, the lead developer was too busy to validate the findings.

Finally, the timing of component reuse incidents and statistics on the developers who performed the reuse were collected for all component and function reuses in the component reuse inventory. In the next section, we turn to the findings.

4. Findings

The aim of this section is to shed light on the research questions developed in Section 2. While we mainly report on code reuse, we also sustain the notion that knowledge reuse in software development covers reuse of problem-solving as well as code (Barnes and Bollinger, 1991, Section 2) and, thus, include other forms of knowledge reuse where appropriate. First, an inventory of the projects studied shows that developers predominantly reuse software components. Second, the research questions are explored and contrasted with the analysis of component reuse and the interview data in order to answer why reuse happens.

4.1 Knowledge and code reuse

There were three broad forms of knowledge and code reuse in the core sample: algorithms and methods, single lines of code, and components. First, an algorithm is a finite set of well-defined instructions for accomplishing some task or solving some problem which, given an initial state, will result in a corresponding recognizable end-state (adapted from wikipedia.org, 2004). Methods contain several alternative algorithms and other scripts for solving a problem, rules for choosing between them, and they can be expanded to cover a large problem area. The reuse of algorithms and methods includes the examination of source code or other information, but also the interpretation and adaptation of cues about technical problems and their solutions, abstraction, and implementation in a local context. Nearly all of the developers interviewed mentioned the reuse of algorithms and methods in their open source software development. The analysis across the core sample revealed that knowledge reuse in the form of methods and algorithms is frequent but rarely credited. Interviews revealed that developers spend non-negligible amounts of time studying scientific publications (such as engineering journals) and standard specifications, or learning from the source code (and its documentation) of related projects. The reuse of algorithms and methods not accompanied by software code reuse is impossible to document completely, because it is part of the individual's learning and usually not made explicit.

Second, copying specific lines of code from external projects is a systematic and direct form of code reuse in open source software development. The analysis of imported "accredited lines of code" amounted to 38,245 across the core sample. For reasons described in Section 3, this form of reuse was relatively rare, hard to quantify, and not further pursued in the current study.

Third, all the core sample projects reused software components. A software component is a software technology for encapsulating software functionality, often in the form of objects, adhering to some interface description and providing an API, so that the component may exist autonomously from other components on a computer. Technically, this autonomy allows the developer to treat the component as a "black box." The components were either integrated into the code of the project or linked to it. Linking a component to the software could happen either at the time of compilation (static linking) or at run-time (dynamic linking). Reuse (or acquisition) of components can be "black-box" or "white-box" (Ravichandran and Rothenberger, 2003), depending on whether changes were made to the reused code. With very few exceptions, the reuse in this sample amounted to black-box reuse since the components were reused without modifications. Across the core sample, 55 components were reused, representing a total of 16.9 million component LOC, while the total LOC of the core sample was 6.0 million (not including the reused LOC). A complete list of reused components can be found in Table 2. Each identified component reuse incident is listed together with a minimal description and its date of reuse. All the components reused in the core sample are maintained as external projects which means that they are available through a dedicated project website, provide code releases or open development, or all of the above.

A closer analysis of the components revealed two distinct types of code reuse: architectural reuse and functional reuse. In order to make use of components that are not developed inside the project's community of developers, a developer has to first search for and integrate a suitable component. The decision to reuse a component introduces an architectural change to the software because it changes its overall structure (Baldwin and Clark, 2000; Ulrich, 1995). According to the developers (B, H, M, L), the decision to reuse a component is based on the functions this component offers. In order to make the code reuse decision, the developers studied not only the software code but also the documentation accompanying the code, web pages with frequently asked questions (FAQs), overview documents and similar web-based resources before deciding to reuse a component. The first step of reuse is then the inclusion of the component in the software. The core sample contained 200 instances of *architectural reuse* which import a component or part of a component. In Table 2, they are attributed to each of the 55 reused components in the fourth column.

The second type of component reuse was termed *functional* and based on previous architectural reuse. Each component offers a number of functions that may or may not be used by the originating program. Through architectural reuse of a component, the developer makes functionality available to the program. The actual use of some of these functions executes the options inherent in the component. Functional reuse incidents became visible through fine-grained analysis of the code base of the sample that revealed each available function. Only the first call of a new function was recorded in order to determine whether it was using functionality provided by components. The core sample contained 2,775 functional reuse incidents. The functional reuse incidents correspond to a reused component and are listed in the fifth column of Table 2.

	Component	Component Description	Reuse date	Reuses	
				architectural	functional
Abiword	libpng	Display 'png' graphics	1999-02-16	1	26
	glib-gtk-gdk	Graphical toolkit	1998-07-28	24	542
	fribidi	Display fonts (bidirectional)	2001-09-13	5	26
	zlib	Compression utility	1999-02-16	1	10
	Libxml	Xml file parsing	2000-07-27	3	20
	enchant	Spell checking library wrapper	2003-07-13	1	11
	ispell	Spell checker	1998-12-28	2	18
	Libwv/libmswordview	View MS Word files	1998-08-28	9	180
	libiconv	unicode	2000-02-01	1	2
	expat	Xml parser	1998-08-31	1	14
	libglade	Graphical toolkit construction	2000-10-11	Indirect through gtk	5
	libpopt	Parsing command line options	1999-03-11	2	11
	libfreetype	Display fonts	2002-05-12	5	8
	libwmf	Display 'wmf' graphics	2001-09-28	3	5
	libjpeg	Display 'jpg' graphics	2001-04-26	1	7
	libcurl	Download files	2002-04-30	2	4
	libgal	Gnome accessibility functions	2000-12-16	6	34
	pango	Display internationalized text	2002-05-05	15	126
GNUnet	openssl	Encryption	2001-09-05	10	28
	libgcrypt	Alternative encryption lib	2003-01-15	8	219
	libpopt	Parsing command line options	2002-01-06	1	3
	gdbm	Data base	2002-05-05	1	5
	tdb	Alternative data base	2002-05-14	1	8
	mysql	Alternative data base	2003-03-01	1	15
	gtk/gdk/glib	Graphical toolkit	2002-03-09	1	160
	libglade	Graphical toolkit construction	2003-10-28	Indirect through gtk	4
	zlib	Compression utility	2001-08-30	1	2
	sleepycat db	Alternative data base	2003-03-30	1	1
irate	javalayer	Mp3 player	2003-03-26	1	54
	xerces	Xml parser	2003-03-26	3	58
	nanoxml	Xml parser	2003-05-26	1	9
	swt libraries	Graphical toolkit	2003-07-24	3	11
	httpClient	Download files	2003-08-06	2	3
	libmadplayer	Mp3 player	2003-03-30	3	3
OpenSSL	zlib	Compression utility	2000-11-30	1	5
TikiWiki	Pear::DB	Mysql wrapper for php language	2003-04-09	1	N/A(mysql)
	mysql	Data base	2002-10-08	Via Pear::DB	15
	Smarty	Php template engine	2002-10-08	1	38
	hawhaw	Provide 'wap' version	2003-05-15	1	6
	htmlarea	Interactive html editor	2003-04-23	2	2
	JGraphPad	Java applet paint program	2003-04-24	1	1
	overlib	Display browser tooltips	2002-11-25	1	1
	jscalendar	Display calender in web browser	2002-10-08	1	1
	adodb	Data base abstraction layer	2003-07-15	3	5
	GraphViz	Generating graph layouts	2003-04-02	1	1
	php-pdf	Convert files into pdf files	2002-10-08	2	18
	phplayers	Php menu system	2003-11-25	1	1
	wollabot	Generic IRC chat bot	2003-11-15	1	1
xfce4	gtk/gdk/glib	Graphical toolkit	2001-02-14	37	863
	libiconv	Font conversion to/from unicode	2002-12-15	1	1
	libxml2	Xml file parsing	2003-02-04	4	21
	alsa	Linux sound library	2003-04-26	4	43
	oroborus	Window manager	2002-05-03	6	63
	libwnck	Window Navigator Construction Kit	2002-10-18	10	40
	pango	Display internationalized text	2002-05-03	1	17
			SUM:	**200**	**2775**

Table 2: Component reuse inventory in the core sample

4.2 Substituting the corporate reuse program

The component reuse inventory demonstrates that open source software developers routinely and widely reuse software components across the sample. The following analysis explores questions 1a through 1c in order to explore how the context of open source software provides equivalent mechanisms that corporate reuse programs feature, namely lower search costs, establishment of quality standards, and the provision of incentives.

The perceived costs to a developer of searching and integrating a new component must stay below the effort of writing software from scratch (Banker et al., 1993). Tools and standards facilitate the search and integration of existing components (Ravichandran, 1999), and reuse in open source software development should only be expected if equivalents to corporate tools and standards exist. Internal search repositories, a solution proposed by Banker et al. (1993), could not be found in the core sample. However, a few large repositories of open source software projects (Sourceforge.net, Savanna, Berlios, etc.) as well as dedicated index and search tools (e.g., Freshmeat, koders.com) offer free infrastructure for projects of various domains and target both end-users as well as developers. Distributions, such as Debian GNU/Linux, publish extensive information that helps developers identify components and dependencies of components which they contemplate using[9]. We found that 85% of all reused components in the core sample were listed in the Debian package repository.

However, even more important than repositories and search engines were means of local search in a known space (March, 1991). Several developers (D, K, F, H) underscored the importance of their respective software community for finding relevant knowledge and code. One developer (F) suggested that of all sources, his project's developers and mailing list participants were the most direct and efficient source of information about reusable knowledge and code. Developer (H) suggested:

---

[9]Dependencies are underlying components that a software requires in order to work. For the Debian package list, that includes dependency information, go to: http://www.debian.org/distrib/packages

> *"I'll post on the mailing list, - the developer list. I just ask everybody: does anyone know about this? Can anyone recommend a good library? Chances are somebody uses one. Might not even be writing code, but they might know something about it."*

Next, standards provided a means of lowering search and integration costs. In the core sample, standards involved stable and documented interfaces to the component functionality, as well as its accessibility in the project programming language, and influenced the developers' decision to reuse. By using a well defined and designed set of variables and commands (API), the developers could access the component's functionality, thus reducing the effort to understand the component (Developers A, G, H, D). With a good interface available, the developers did not have to fully understand the inner technical workings of the component in order to be able to use it and they could integrate the component into the software more easily. Accessibility of the component to the project's programming language also proved to be important. For example, the developers of iRATE radio considered replacing a component with another external library (libmadplayer) in order to take advantage of its advanced functionality. However, a lack of compatibility between the two programming languages Java and C made this difficult:

> *"But if we switch to libmad[player], we'll have to maintain the Java interface from libmad to iRATE. [...] They don't have a Java interface right now. Basically it's to use libmad, it's written in C, and [it is] sort of difficult to interface C with Java code. Whereas [for] Madplay, - it's just a program [front end for libmadplayer] and Java doesn't really care what program it is as long as it can run from command line. But [with] libmad, we'll actually have to call internal routines, - it's much more difficult to keep it up to date." (Developer I)*

The risk of unforeseen changes within reused components was mediated by what developers considered "a wide consensus" among open source software projects to guarantee API stability within major releases. As developer D states, the effort of integrating a component increases when the interfaces change too often:

”If [the component] changes constantly and it's incompatible with your project, it causes overhead. You don't want to have 50 million conditionals for every software that exists. In this case you might choose to copy over their code or email the software's maintainers and say; “listen, we're trying to use your code but you keep changing it on us, is there any way you can keep this stable?” Frequently, they will be happy to comply because they'll have extra users of their code that help find bugs. There are several options, it's good to know the policy but it's not necessary for the initial use of their code.”

The interviews revealed that open source software development offers equivalents to search tools and standards which facilitate the developers' search for and integration of components, positively replying to question 1a.

Knight and Dunn (1998) point out that developers in firms are unlikely to reuse a component unless they can trust its quality, since an external component can potentially harm the overall system. Therefore, they propose certifying components for reuse. In the core sample, the “popularity” of a component served as a substitute to certification and generally signaled the quality of the software (developers H, G, F). The developers reasoned that bug fixes in the software are more frequent in widely used components leading to better quality:

“There's a higher probability that more people have looked at the code, figured out if it works, how it works, and probably fixed more bugs if there are any. It's the whole peer review thing: the more people have looked at the code and still use it, the more it's trustable.” (Developer F)

A straightforward indication of the popularity of a component is its inclusion in a major software distribution such as Debian which is peer reviewed, actively maintained, and reaches a wide user base. (As mentioned above, 85% of the reused components in the inventory were included in the Debian distribution). This answers question 1b: by evaluating the popularity of components, open source software developers indeed use a proxy to certification and quality standards in order to support code reuse.

A software development firm needs to create incentives for developers to reuse code because they generally perceive reuse to be less rewarding than writing new code. Von Hippel and von Krogh (2003) argue that, in the private-collective model of innovation, developers expend resources privately to contribute to a public good. If the developers perceive their resources as limited, reuse could help to mitigate development costs. In this case, the net savings in development costs through reuse should act as individual incentives. Support was found for this conjecture: developers in the core sample explicitly reused code in order to reduce their costs of creating the software. They spent available resources in consideration of both available time and skills (developers K, D, H, A):

> *"The primary driver of that [reuse] decision is making the project the best it could be. The fact is that we're mortal and we don't have an infinite amount of time to rewrite everything. So even if the other project's code isn't perfect but good enough, you're simply going to use it because if you've got a thousand bugs to fix you don't want to spend the next year rewriting all the software you could just use from someone else." (Developer D)*

The interviews confirmed the existence of resource constraints in open source software development. The economic rationale of saving development costs, where possible, was consistent among developers' replies. The need to advance the code base and approach the objectives of the overall project was weighed against the time and skills available and influenced the decision to reuse code, as shown in this example:

> *"There were cases where it was better to have a third-party program. OK, for example jgraphpad, which is a java applet, to have graphics. We wouldn't necessarily have the expertise on the team to do that, or the time or the interest, but that was a perfect match between what they were doing and what we were doing." (Developer K)*

An additional incentive to reuse was the option to outsource the maintenance work through reuse. For 53 out of 55 reused components, at least one new release became available after the first date of reuse. Hence, the reusing projects could benefit from "free" maintenance by other projects. Developers (B, K, H) considered external maintenance as an incentive to reuse components because it lowered the long-term costs of producing a component inside the community by the effort to maintain it, particularly regarding internal bugs and errors. The developers in our sample systematically reused components because, first, they saved effort by not having to write the component, and second, by not having to maintain it in the future. These findings relate to question 1c; developers' limited time and skills create incentives to seek savings in development costs through code reuse. The pattern that an economic logic influences reuse behavior also positively replies to question 2c.

4.3 Fulfilling the credible promise

The credible promise, or the release of workable software that is complete enough to work on, helps attract users and potential developers to a project (as described in von Krogh et al., 2003). Accordingly, this section explores questions 2a through 2c. One way to quickly establish a working code base is to integrate existing components and building on existing functionality. The findings shed light on question 2a: developers reused components as early as of day one of the project's inception. Figure 1 shows that 666 of 2,975 reuse incidents (22%) already occurred during the first 10% of the observation period for the core sample projects. The credible promise is fulfilled in the core sample: the first public software release happened on average already 44.5 days after a project's inception[10].

[10]The day of inception is the first day on which code is added to the repository. It is possible that part of the source code existed earlier outside of the repository.

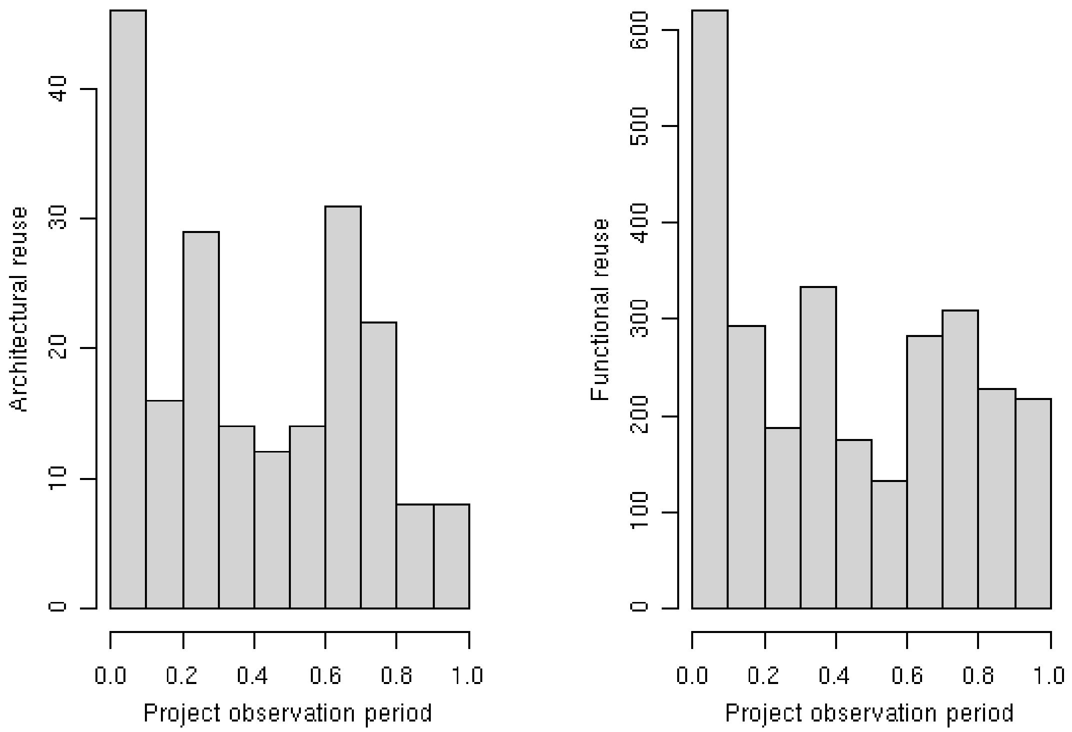

Figure 1: Reuse incidents over the total observation period

The patterns of architectural reuse resemble those of functional reuse; new components (or parts of components) were added to the code base throughout the observation period. Hence, developers make use of the (modular) design options by adding components. This observation is consistent with a claim made in the literature that developers exercise design options in software architecture (Baldwin and Clark, 2006).

As mentioned above, developers chose areas they like to work on through self-assignment of tasks. But since an open source software project also requires the execution of mundane and difficult work for developers, question 2b asks if code reuse could help to evade the writing of “mundane” code and focus on more rewarding programming tasks specific to the project and that fit the developers’ skills.

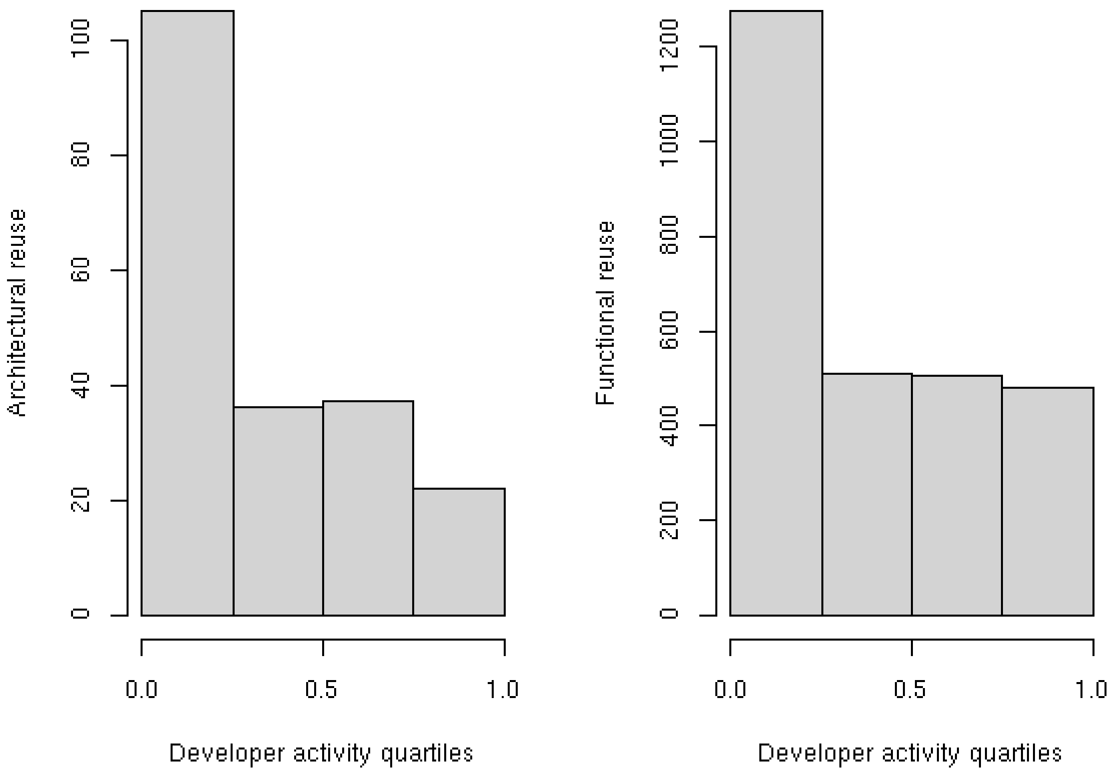

Figure 2: Reuse incidents during the developers' "life span"[11]

The developers' individual reuse behavior shows increased reuse early on in the developers' coding activity for the project (Figure 2). The total median in this sample (architectural and functional reuse combined) was 0.28 (avg: 0.37, sd: 0.32). Thus, on average, developers performed reuse after having contributed a third of their total number of lines of code[12]. This implies that most developers remained active after a reuse incident and continued to write code for the project. The interviewees confirmed the developers' preference for reusing early during their active period on the project. They (J, H, K, A, E) perceived reuse as an opportunity to get rid of mundane, time-consuming, or difficult coding tasks, that helped them to work on their preferred tasks. Developer E sums it up:

[11]The developers' "life span," that is their total coding activity over time, was divided into quartiles. The reuse incidents were allocated to the individual activity quartiles in order to visualize the reuse incidents over the developers' "life spans." The histogram shows when reuse incidents happened for every quartile. "0" on the activity axis indicates a reuse incident as the first activity (in LOC), "1" a reuse incident as the last observed LOC written by the developer.

[12]The data also showed that long-term and more active developers performed more code reuse, not in relative, but in absolute terms. The reason may be that these developers have acquired a better familiarity with the code base. We are grateful to the Associate Editor for pointing this out. Dividing developers into two groups (contributing for more/less than 50% of the observation period preceding the reuse incident), additional analysis showed no significant differences between the groups in code reuse frequency over a developer's life span in a project.

> *"Code reuse is just helping us to get the job done, so I can work on something that is more interesting."*

These findings answer question 2b: software reuse helped developers to get mundane or difficult tasks done and allowed them to focus on "interesting" (preferred) areas of work.

Finally, as elaborated in subsection 4.2, resource constraints were explicitly and frequently mentioned as reasons for code reuse by developers. The developers benefited from "free" maintenance and improvements made to project-external components and chose the least costly path to ensure that workable code could be released and progress was being made. This behavior relates to the finding that developers spend their scarce resources economically. Component reuse helped to advance the project, thus answering question 2c positively.

5. Conclusion and implications

In the "private-collective" innovation model, the benefits must outweigh the cost of contributing to the public good (von Hippel and von Krogh, 2003). Knowledge reuse can be a strategy to mitigate the costs of innovation (Langlois, 1999) and commercial software engineering practices emphasize the reduction of developments through code reuse (e.g., Barnes, et al. 1987; Barnes and Bollinger, 1991; Banker and Kaufmann, 1991). This study departs from two contradicting issues, namely that open source software licenses are designed to enable and encourage sharing and building on others' work, yet reuse is hard to achieve in commercial settings. It shows that developers in open source software projects actively reuse available code and other knowledge that solve their technical problems, and it presents empirical evidence on the extent of code reuse, and the development practices of developers in open source software projects where resources are scarce, highlighting the importance of reusable software components. In the sample of six open source projects, this research identified 55 reused components comprising 2,975 reuse incidents. The findings are presented against the back drop of the literature on code reuse in software development firms, offering insights with regard to the context of open source software. The data from the core sample showed that developers used tools and relied on standards when reusing components. They used "popularity" as a proxy for quality ratings and acted under resource constraints (time and skills) when selecting reusable components. While core developers participated in the reuse activities, they offered no explicit encouragement for code reuse across their project. As predicted by the existing literature on open source software development, building an initial credible promise required code reuse early on during the project, with developers continuing to reuse code as resource constraints persisted and suitable open source components were available. The developers continued to reuse and stay active after their initial, extensive reuse behavior. This finding is consistent with Shah (2006) who showed that the type of tasks tackled by long-term developers changes over time. In summary, the developers reused for three reasons: they wanted to integrate functionality quickly (first public release after 44.5 days on average), they preferred to write certain parts of the code over others, and they could mitigate their development costs through code reuse.

Implications for research

Most component reuse in open source software development crosses project boundaries, thereby enlarging the project resources by effectively outsourcing part of the development. In particular, these additional resources might help young projects to gain the necessary momentum to reach a critical mass. This represents an alternative strategy to community growth and resource mobilization (e.g., Bonaccorsi and Rossi, 2003). Future research should examine this form of growth in more detail.

The software reuse literature (Tracz, 1995) estimates that the cost of building reusable components adds up to 200% of *additional* development costs. Future research should uncover who carries these costs in open source software development and why. As this study has shown, repositories such as Sourceforge offer vast amounts of reusable software and most components were released by projects dedicated to that specific component development. Casual evidence from the core sample suggests two possible explanations: the developers' mobility across software projects (proprietary and open source) benefits from reusing parts of their own work in parallel or subsequent projects. Second, reputation rewards from peers for “clean”, that is modular and well structured, code justify the private, additional efforts to build reusable components. Both arguments deserve further in-depth examination.

Certain projects reuse more code than others, but the antecedents of reuse in open source software development are largely unknown. Future research should identify individual and organizational characteristics that impact on reuse. Open source licenses enable the “outsourcing” of functionality to other projects. The resulting shared use of code across projects may be related to the total level of reuse and the cost of innovation. Future research needs to identify the factors that drive such outsourcing behavior.

As table 2 shows, certain components are reused more frequently than others. Future research should analyze the characteristics of reused components in order to predict the frequency of component reuse. This study should inform the design for reuse. The results from the present study pertaining to the search and maintenance costs, and the trust in components should inform hypotheses about component characteristics that lead to high reuse frequencies. Future advancement of the research on the above research issues should be built on a categorization of components in terms of functionality and, possibly, quality in order to theorize about reuse success.

Finally, we found that long-term and more active developers performed more code reuse in absolute, but not in relative terms (see footnote 12). The controls showed that the relative pattern of reuse activity over the total time of coding activity remained unchanged across developers who had coded for the project for more or less than 50% of the observation period preceding the reuse incident. Future research needs to investigate if the cause of this is individual developer learning, familiarity with the code base, stronger specialization in the functionality of the software, or other reasons.

At least two limitations apply to this research. First, the study focused on code reuse, specifically component reuse. Data on single accredited lines of code was based on interviews and on an automatic analysis. By definition, the reuse of these lines of code requires developers to credit those who wrote these lines in the code modification comment made in their own projects. Considering the sheer amount of existing open source projects, non-credited lines of code reuse is close to impossible to identify. Therefore, the correctness of the accredited lines of code-analysis cannot be guaranteed. Yet, the developers stated that giving credit for reused code is a social norm when sharing code across projects (see also Fauchart and von Hippel, 2006). Future research into code reuse might need to capture accredited lines of code reuse through survey data in addition to components.

Second, the initial sampling only included projects in active development. Since active development enables the observation of code reuse practice in real time, the analysis could combine interview material with current examples and easy-to-verify component origins (e.g., whether the component was under active development in another project). Defunct projects were excluded. In order to understand the full performance implications of code reuse, future research should compare code reuse in successful and failed projects.

Implications for management practice

Open source software projects offer a vast repository of readily usable software for almost all purposes. Within the limits of the licenses and the mechanisms that communities apply to protect their work (O'Mahony, 2003), this software can be used, reused, and built upon freely (see also Henkel, 2006). This corresponds to the advantages of black-box reuse in component markets (Ravichendran and Rothenberger, 2003) with the difference that the open source components are available for free. Commercial software developers may observe and learn from open source software developers' work. The available repository of knowledge and code could lower the probability of reinventing the wheel for firms and communities by offering methods and algorithms from open source solutions for reuse. Thus, managers of software firms should encourage and support the learning process for developers who spend time looking at available open source software.

Managers who allocate developer resources to open source software projects will possibly see new practices of innovation develop in their firms. Developers exposed to open source software development might bring practices into the firm that favor knowledge reuse over the reinvention of the wheel and introduce elements of an open source software development culture that could change the firm's internal culture.

An organization-wide, corporate reuse program is not a prerequisite for code reuse. This is an important lesson for management practice. Information about the popularity of software may substitute a costly certification process and enhance the developer's trust in the code. This study also showed that strong incentives for code reuse exist if the software developers act as "software entrepreneurs". Software developers who are compensated for task achievement, rather than time spent, have incentives to cut development costs and to reuse existing functionality. This *could* imply that if a software firm creates an entrepreneurial organization for its software developers, it may complement the importance of other reward-based incentives.

The equivalent to incentives in corporate reuse programs in the context of open source software development could help managers structure more effective non-monetary incentives. Recognition and extrinsic awards were found to promote code reuse in firms (Poulin, 1995; Isoda, 1995). In open source software, the continuous maintenance of reused components by others created the perception of free maintenance for the reusing developers. Avoiding to write a component from scratch combined with the free (external) maintenance provides incentive for reuse. Possibly, managers allowing developers to self-assign tasks may achieve the necessary level of component maintenance to encourage reuse and can further facilitate reuse by separating the maintenance and reuse of existing components.

When inspecting the component reuse inventory, established and well-known low-level components can be identified, such as encryption software (OpenSSL), compression software (zlib), databases (MySQL), or graphical toolkits (GTK). These have all proven useful to a large audience of users and developers. The reuse behavior of open source software developers also informs the potential commercial suppliers of software components about the structure of this “emerging market”. Similarly, experience from corporate component reuse shows that domain-independent reuse (often low-level components) is easier than domain-specific reuse due to lower adaptation costs (Poulin, 1995; Ravichandran and Rothenberger, 2003).

Interviews revealed that open source software developers work under severe time and skill constraints. The self-inflicted pressure to release a working product leads to efficiency thinking and economic behavior with regards to the utilization of scarce resources. The insights from innovation process research in open source software continue more than ever to be useful to researchers studying innovation in firms (particularly software firms), since similar economics of innovation apply in both contexts. The limitations of researching into the alleged "hobbyist culture" of open source software (Carbon et al., 2001; Moody, 2001) does not apply when studying the social and technical processes of innovation in open source software development. As shown, there are important lessons for researchers and managers considering innovation in both contexts. Work on open source software development and its commercial counterpart will mutually benefit from an exchange of results and jointly they can contribute to an extended theory and insights into private-collective innovation.

6. References

Argote, L. 1999. Organizational learning: Creating, retaining, and transferring knowledge. Kluwer, Norwell, MA.

Apte, U., C.S. Sankar, M. Thakur, J.E. Turner. 1990. Reusability-based strategy for development of information systems: Implementation experience of a bank. *MIS Quarterly.* December. 375-401.

Baldwin, C.Y., K.B. Clark. 2000. *Design Rules, Volume 1, The Power of Modularity*. MIT Press.

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

Banker, R.D., R.J. Kauffman. 1991. Reuse and productivity in integrated computer-aided software engineering: An empirical study. *MIS Quarterly.* **15(**3) 375-401.

Banker, R.D., R.J. Kauffman, D. Zweig. 1993. Repository evaluation of software reuse. *IEEE Transactionson Software Engineering*. **19**(4) 379-389.

Barnes, B.C., T.B. Bollinger. 1991. Making reuse cost-effective. *IEEE Software.* **8**(1) 13-24.

Barnes, T., T. Durek, J. Gaffney, A. Pyster. 1987. A framework and economic foundation for software reuse. *Proc. Workshop Software Reuse and Maintainability*.

Basili, V.R. 1990. Viewing maintenance as reuse-oriented software development. *IEEE Software*. **1** 19-25.

Benkler, Y. 2002. Coase's Penguin, or Linux and the Nature of the Firm. *Yale Law Journal*. **112**(3).

Bonaccorsi A., C. Rossi. 2003. Why Open Source software can succeed. *Research Policy*. **32**(7) 1243-1258.

Carbon, G., A. Lesniak, D. Stoddard. 2001. *Open Source Enterprise Solutions: Developing an E-Business Strategy*. John Wiley & Sons, New York.

Cusumano, M. 1991. *Japan's software factories*. Oxford University Press.

Cybulski, J.J., R.D. Neal, A. Kram, J.C. Allen. 1998. Reuse of early life-cycle artifacts: work products and methods, and tools. *Annals of Software Engineering*. **5** 227-251.

Fauchart, E., E. von Hippel. 2006. *Norms-based intellectual property systems: The case of French chefs.* MIT Sloan School of Management Working Paper 4576-06.

Favaro, J.M., K.R. Favaro, P.F. Favaro. 1998. Value-based software reuse investment. *Annals of Software Engineering*. **5** 5-52.

Frakes, W., S. Isoda. 1994. Success factors of systematic reuse. *IEEE Software.* **11**(5) 15-19.

Frakes, W.B., C. Terry. 1996. Software reuse: Metrics and models. *ACM Computing Surveys*. **28**(2) 415-35.

Gamma, E., R. Helm, R. Johnson, J. Vlissides. 1995. *Design Patterns*. Addison-Wesley.

Garud, R., A. Kumaraswamy. 1995. Technological and organizational designs for realizing economies of substitution. *Strategic Management Journal*. **16** 93-109.

Griss, M.L. 1993. Software reuse: From library to factory. *IBM Systems Journal*. **32**(4) 548-566.

Henkel, J. 2006. Selective revealing in open innovation processes: The case of embedded Linux. *Research Policy*, **35** 953-969.

Hertel, G., S. Niedner, S. Herrmann. 2003. Motivation of software developers in Open Source projects: an Internet-based survey of contributors to the Linux kernel. *Research Policy*. **32**(7) 1159-1177.

Isakowitz, T., R.J. Kauffman. 1996. Supporting search for reusable software objects. *IEEE Transactions on Software Engineering*. **22**(6) 407-423.

Isoda, S. 1995. Experiences of a software reuse project. *Journal of Systems Software*. **30** 171-186.

Joos, R. 1994. Software reuse at Motorola. *IEEE Software.* **11**(9) 42-47.

Kim, Y., E.A. Stohr. 1998. Software reuse: survey and research directions. *Journal of Management Information Systems*. **14**(4) 113-147.

Knight, J.C., M.F. Dunn. 1998. Software quality through domain-driven certification. *Annals of Software Engineering*. **5** 293-315.

Lakhani, K.R., R.G. Wolf. 2005. Why hackers do what they do: Understanding Motivation and effort in Free/Open Source software projects. In: Feller, J. Fitzgerald, B., Hissam S., K.R. Lakhani (eds.). *Perspectives on Free and Open Source software.* MIT Press.

Langlois, R.N. 1999. "Scale, scope, and the reuse of knowledge" in Dow, S.C., P.E. Earl (Eds). *Economic Organization and Economic Knowledge*. Edward Elgar, Cheltenham. 239-254.

Lerner J., J. Tirole. 2002. Some Simple Economics of Open Source. *Journal of Industrial Economics*. **52**.

Lim, W.C. 1994. Effects of reuse on quality, productivity, and economics. *IEEE Software.* **11**(9) 23-30.

Lynex, A., P.J. Layzell. 1998. Organisational considerations for software reuse. *Annals of Software Engineering*. **5** 105-124.

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

Majchrak, A., L.P. Cooper, O.P. Neece. 2004. Knowledge reuse for innovation. *Management Science*. **50**(2) 174-188.

March, J.G. 1991. Exploration and exploitation in organizational learning. *Organization Science*. **2** 71-87.

Markus, M.L. 2001. Towards a theory of knowledge reuse: Types of knowledge reuse situations and factors in reuse success. *Journal of Management Information Systems*. **18**(1) 57-93.

McClure, C. 2001. *Software reuse: A standards-based guide*. IEEE Computer Society.

Moody, G. 2001. *Rebel code*. Perseus Publishing, Cambridge MA.

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

Prieto-Diaz, R. 1993. Status report: Software reusability. *IEEE Software*. **10**(3) 61-66.

Poulin, J.S. 1995. Populating software repositories: Incentives and domain-specific software. *Journal of Systems Software*. **30** 187-199.

Ravichandran, T. 1999. Software reusability as synchronous innovation: A test of four theoretical models. *European Journal of Information Systems*. **8** 183-199.

Ravichandran, T., M.A. Rothenberger. 2003. Software Reuse Strategies and Component Markets. *Communications of the ACM*. **46**(8) 109-114.

Raymond, E.S. 2000. *The cathedral and the bazaar: Musings on Linux and Open Source by an accidental revolutionary*. Version 3.0. URL:www.catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/cathedral-bazaar.ps.

Rothenberger, M.A., K.J. Dooley, U.R. Kulkarni, N. Nada. 2003. Strategies for Software Reuse: A Principal Component Analysis of Reuse Practices. *IEEE Transactions on Software Engineering*, **29**(9) 825-837.

Schmidt, D.C., M. Fayad, R.E. Johnson. 1996. Introduction. *Communications of the ACM*. **39**(10) 36-39.

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

Stake, R.E. 1995. *The art of case study research.* Sage.

Tracz, W. 1995. *Confessions of a used program salesman: Instiutionalizing software reuse*. Addison-Wesley.

Ulrich, K. 1995. The Role of Product Architecture in the Manufacturing Firm. *Research Policy*. **24** 419-440.

von Hippel, E. 2001. User toolkits for innovation. *Journal of Product Innovation Management*. **18**(4) 247-257.

von Hippel, E., G. von Krogh. 2003. The Private-Collective Innovation Model in Open Source Software Development: Issues for organization science. *Organization Science*. **14**(2) 209-223.

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

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

von Krogh, G., E., von Hippel. 2006. The promise of research on Open Source software. *Management Science*. **52**(7) 975-983.

Yamauchi, Y., M. Yokozawa, T. Shinohara, T. Ishida. 2000. Collaboration with Lean Media: How Open-Source Software Succeeds. *ACM Conference on Computer Supported Cooperative Work*. Philadelphia, PA.

Yin, R.K. 1989 *Case study research:* Design *and methods*. Sage.

Zander, U., B. Kogut. 1995. Knowledge and the speed of the transfer and imitation of organizational capability: An empirical test. *Organization Science*. **6**(1) 76-92.