The Knowledge Ecology of Open-Source Software Projects

Giovan Francesco Lanzara* and Michèle Morner**

* Dipartimento di Organizzazione e Sistema Politico
Università di Bologna
Via de' Bersaglieri 6
I – 40125 Bologna
E-Mail: lanzara@spbo.unibo.it

** Wirtschaftswissenschaftliche Fakultät Ingolstadt an der
Katholischen Universität Eichstätt-Ingolstadt
Auf der Schanz 49
D - 85049 Ingolstadt
E-mail: michele.morner@ku-eichstaett.de

Abstract
In this paper we characterize the processes of knowledge making in open-source software projects as an ecology of agents, artifacts, rules, resources, activities, practices and interactions. In order to grasp its dynamic features we consider open-source software projects as interactive systems based on dense interactions between humans and technical artifacts within electronic media. Technology, rather than formal or informal organization, embodies most of the conditions for governance in open-source software projects, hence becoming a critical pathway to the understanding of collective task accomplishment, coordination and knowledge making processes. Based on an in-depth analysis of two open-source software projects, we examine three kinds of artifacts, respectively inscribing technical, organizational, and institutional knowledge. The artifacts are 'dynamic vehicles' that make knowledge creation, accumulation, and dissemination possible by circulating throughout the Internet. For each of them we describe the different activities and the network dynamics in which they are embedded. Our preliminary findings support the ecological view, that the contradictory requirements of innovation and stability in project-based knowledge making are balanced by mechanisms of variation, selection, and stabilization.

Index

1. Introduction

Open-source software projects provide a rich field to explore the processes of creation, accumulation and dissemination of knowledge in spatially dispersed settings where project membership is fluid and participation is volatile. They represent a special example of decentralized "project-based learning" (DeFillipi, 2002; DeFillippi & Arthur, 2001). Differently from standard industry, where software is developed in-house by firms and then sold out as a finished product to customers on the market (Lakhani & von Hippel, 2000; von Hippel & von Krogh 2003), in open-source software projects as many as thousands of skilled programmers and users collectively develop the software online via the Internet in a decentralized, highly interactive, knowledge-intensive, apparently unmanaged process (Raymond, 2001; Lerner & Tirole, 2001; Kogut & Metiu, 2001). Groups of developers and users working in geographically dispersed settings are supported by a dense network of interactions that make them look like communities rather than task-oriented teams or organizations (Brown & Duguid, 1991; Tuomi, 2000; von Krogh, 2002). Developers build the software by relying on extensive peer production and through the skillful use of the software and communication tools available on the Internet. A host of Internet-based dot.com distributors, vendors and brokers package and distribute the software to consumers, but users can also bypass the distributors and download software directly from the web. The source code, which is the basis of the software, is not a trade secret but open for inspection and change by other programmers and users without the legal restrictions typical of commercial software.

We submit that open-source software development poses a theoretical challenge to conventional ways of conceptualizing knowledge processes within and across organizations (see for example the special issue of Strategic Management Journal on 'Knowledge and the Firm' edited by Grant & Spender, 1996; the special issue of Journal of Management Studies on 'Knowledge Management' edited by Swan & Scarbrough, 2001, and the special issue of Organization Science on 'Knowledge, Knowing, and Organizations' edited by Grandori, Kogut & Lewin, 2002; furthermore Eisenhardt & Santos, 2000; Nonaka & Takeuchi, 1995; Patriotta, 2003). Despite their loosely

coupled organization, open-source software projects are surprisingly effective in tapping and coordinating the talents of many spatially dispersed contributors (Raymond, 2001; von Hippel, 2001; Kogut & Metiu, 2001; Metiu & Kogut, 2001). Distributed knowledge resources of the most diverse kinds are gathered from multiple sources, productively used to develop code, and swiftly circulated across geographical and organizational boundaries (Kogut & Metiu, 2001). Open-source software incorporates the demands and the practical knowledge that the users themselves develop in designing and testing software online (Neff & Stark, 2002). Innovation happens through design-in-use by what von Hippel refers to as 'user communities' (von Hippel, 2001). Knowledge creation and dissemination take place in a fast and cost-effective way, challenging conventional views that see the market (von Hayek, 1945) or the corporation (Kogut & Zander, 1992; Metiu & Kogut, 2001) as superior vehicles for creating and transferring knowledge. In open-source software projects knowledge making takes place mostly in an electronic media, and the specific features of the media make a difference for the process. Though open-source software projects are increasingly mentioned as a new organizational model for merging and disseminating knowledge resources (Kogut & Metiu, 2001; von Krogh, Spaeth & Lakhani, 2003) and for producing innovation (von Krogh & von Hippel, 2003), its knowledge dynamics and its underlying mechanisms have not been fully tracked yet. This paper is a contribution to the understanding of the knowledge dynamics taking place in open-source software environments. In a nutshell, our point is that open-source software projects are evolutionary systems based on dense interactions between humans and technical artifacts within an electronic media. In such an environment knowledge processes develop by means of variation, selection, and stabilization, gaining a distinctive ecological quality.

Our inquiry is based on three interconnected arguments. First, we argue that open-source software projects should be conceptualized as 'interactive systems'. In order to capture the essence of knowledge making in open-source software projects, we shift the theoretical and empirical focus from organization as an identifiable entity to ongoing interaction as a primary 'organizing'

phenomenon. Knowledge is the emergent outcome of interaction; it is a communication-based phenomenon.

Second, we argue that if we want to grasp how knowledge creation and dissemination happen in open-source software projects we should look at the technology. Interactive systems are deeply intertwined with a web of artifacts and tools that support programming, development and knowledge making activities at large. Technology, rather than formal or informal organization, embodies most of the rules and conditions for governance in open-source software projects, hence becoming a critical pathway to the understanding of collective task accomplishment, coordination and knowledge making processes.

Third, we submit that the diversity of knowledge processes and practices in open-source software projects will be better appreciated if we focus on their distinctive ecological quality. Open-source software projects are intricate webs of agents, practices, artifacts, tools, resources, problems and solutions that co-exist and interact in an ever evolving ecology, where the evolutionary mechanisms of variation, selection, and stabilization keep the activity system in a dynamic balance. Following Bateson (1972), we stress the importance of relations over elementary parts in knowledge systems. In an ecological perspective, knowledge is created by leveraging the scattered and occasional contributions of many small agents. Knowledge is then the unplanned, evolutionary outcome of the complex interplay of a broad variety of heterogeneous elements rather than being a manufactured commodity or the product of smooth conversion processes.

As a methodological choice, we place artifacts and their critical role as dynamic holders and vehicles of knowledge in the focus of our observations. Artifacts play the role of media through which humans communicate and act. Empirically, we look at how artifacts are affected by and in turn support the mechanisms of variation, selection and stabilization in the making of technical, organizational, and institutional knowledge. For 'technical' knowledge we analyse how new versions of the code are generated, updated, and stabilized; for 'organizational' knowledge we analyse the structure of discussion threads found in the projects' mailing lists; for 'institutional'

knowledge we analyse how licencing agreements induce and protect institutional environments that enhance both stability and innovation. We have chosen an explorative approach based on the in-depth analysis of two large-scale open-source software projects, LINUX and APACHE. Our preliminary findings are interesting enough to encourage further research. The paper proceeds as follows: in the next section we articulate the main elements of our theoretical framework. In section 3 we briefly describe our data sources. Then the analysis of how knowledge is made and shared through specific artifacts is developed in section 4. For each chosen artifact we illustrate how the mechanisms of variation, selection and stabilization work. In the final section we offer a synthetic discussion and some concluding remarks.

2. Towards a theoretical framework: interaction, technology, and the ecology of knowledge

Our strategy of inquiry is based upon a shift of theoretical attention from organizations to interactive systems, from organizational routines to technological artifacts, from knowledge making as a manufacturing activity to knowledge making as an ecological process: We characterize open-source software projects as interactive systems based upon computer-mediated interaction (2.1) stressing artifacts as inscriptions of human agency and knowledge (2.2). Knowledge making processes are characterized by the balanced mix and processual interplay of multiple heterogeneous elements, hence retaining a distinctive ecological quality (2.3). In open-source software projects the contradictory requirements of innovation and stability are balanced by mechanisms of variation, selection, and stabilization (2.4).

2.1 Interactive systems

In large scale open-source software development projects dense social connectivity supports a programming environment in which diverse talents and skills are effectively coordinated and put to work on a complex task and in which distributed knowledge resources are tapped for design and innovation purposes (Metiu & Kogut, 2001; von Hippel, 2001). If for once we put aside an organization-theoretic bias, open-source software projects, at their most basic level, can be usefully

regarded as interactive systems (Goffman, 1983; Luhmann, 1995). According to Luhmann (1995: 412-416) interaction refers to communication between those present, and interactive systems emerge when several individuals engage in related communication and perceive the fact that they are engaged in communication. When presence ends, the system ends. However, in open-source software projects participants are not 'present' in a classical (physical) sense, only virtually. They are present in the form of perceiving each other in an electronic medium – the Internet. In principle, they have the possibility to follow each and every communication because of its documentation in mailing lists, bulletin boards or newsgroups websites. In other words, they become 'co-present' when they connect and 'talk' throughout the web from their workstations, otherwise they disappear.

Although not 'organized' in the classic sense, interactive systems have interesting structuring properties. First, they require an attentive core, which consists, for example, in a common subject or task that must be attended to. The subject provides the system with a structure, as the boundaries of the subject regulate participants' contributions (Luhmann 1975 :24), and those who do not attend to or participate in its development cannot influence it. Participation and influence on the subject depend on the amount of attention and time that the agent is able or willing to put on the subject. This functional selectivity, integrating participants' ability of attention and remembering, turns time to structure, i.e. the limited ability to participate essentially creates the structure of the interactive system. Thus, some foci of attention emerge and disappear. Clusters of activity coalesce and disband.

In open-source software projects common subjects or themes are quite typical of communication and development processes. They attract and 'anchor' the attention and skill of the programmers. Attention, being a scarce resource, is intrinsically selective. Agents cannot pay attention to an unlimited number of themes simultaneously, but must have priorities which produce sequences. As a consequence, communication threads are generated. They retain a next-next-next pattern: a message calls for the next around a specific programming task or discussion theme and along an ongoing stream of activity and sense. The close sequencing of communications generates a

self-sustaining process fostering intertemporal connectivity. By directing or diverting attention participants create temporal sequences of entries and exits around a specific thread, sometimes producing what has been called a 'flocking effect', that is, the sudden convergence of attention and people around a theme that happens to be aired. Threads may emerge unexpectedly, then suddenly disappear, but when they reach some stability and persistence over time they play an important part in modularly structuring the project. As we shall see in the next section, the thread pattern stimulates reciprocity and becomes the basis for coordination and knowledge making in open-source software development. Similarly, focal points can be generated (Schelling, 1960; Lanzara, 1997), which have ordering properties in the interaction space: individuals that would otherwise interact randomly arrange and coordinate their interactions depending on the focal point.

Yet, together with the intrinsic self-organizing properties of interactive systems, open-source software projects exhibit some characteristics which are more typical of formal organizations. First, together with fluid and informal participation there is also stable membership, usually limited to a core of professional developers who are in control of critical functionalities or development areas of the project. Second, together with largely non-governed processes, there are some governance mechanisms at work. If for some respects open-source software projects have been regarded as chaotic systems (Kuwabara, 2000), nevertheless most open-source software projects rely upon simple decision making rules, both for programming and communicating. Authority is allocated to make critical decisions in restricted areas. In addition to that, there are also rules governing transactions both within a single project and between the project and its institutional environment. Third, differently from pure interactive systems, open-source software projects are able to build up memory: they keep track of their products and development processes through online, quasi-automatic documentation. Such feature allows for intertemporal travelling throughout the project (going backward and forward), thus helping project continuity, identity and sensemaking (Weick, 1995). Through such processes of continuous tracking and re-tracking, memory becomes a mechanism for the stabilization and self-reproduction of the activity system. Fourth and last, similar

to organizations open-source software projects are equipped with representation mechanisms which allow them to directly communicate with their environment (and be recognized by it), both for symbolic and commercial purposes. Representation can happen in a variety of ways: through a broadly acknowledged leader, a professional team or a steering committee, a sponsoring company, or an institution. The presence of these organizational features makes open-source software projects interesting hybrids. However, they do not really have a dominant or pervasive role in open-source software projects, and taken alone would not be strong enough to account for the impressive performance of large scale projects both as task-oriented production systems and as knowledge making and sharing mechanisms. Most of the organizational gear is invisible in open-source software projects, being draped in the technical gear. In open-source software projects a large quota of 'organization' and organizing is embodied in the technology. Therefore, if one wants to search for organization, he/she'd better look inside the technology.

2.2 Technological inscriptions

It is difficult to have a grip of coordination and knowledge processes in open-source software projects without an understanding of the organizing role of technology (Goodman & Sproull, 1990; Orlikowski, 1992). A number of authors see technology as an inscription of human agency and knowledge (Akrich & Latour, 1992; Latour & Wolgar, 1979; Latour, 1992; Joerges & Czarniawska, 1998; Patriotta & Lanzara, 2001). Following Latour (1992), by *inscription* we mean the act (or sequence of acts) by which humans cast relevant components of their agency and knowledge into artifacts, to which action programs and capabilities are delegated. As a result of delegation, artifacts become holders and dynamic vehicles of human agency: agents 'authorize' artifacts to do things and perform functions at their place or in their behalf in complex networks of human and nonhuman actants (Akrich & Latour, 1992). In open-source software projects technology is an inscription in a two-way fashion: as a communication medium and a software product.

To begin with, open-source software projects live in an electronic medium that confers specific properties to interactive systems and to the ongoing development practices. As the

literature on computer-mediated communication has pointed out (Eveland & Bikson, 1988; Kling & Scacchi, 1982; DeSanctis & Monge, 1998, Kollock, 1999), the medium allows for one-to-all communication, asynchronicity, ubiquity of agents, extended network-based transactions. These features are constitutive of the social practice (and the social order) of open-source. The interactive systems of open-source software projects are largely inscribed in (and supported by) the Internet, the basic information infrastructure. The Internet enhances the social connectivity and facilitates its conversion into purposeful collective action in a very cost-effective way, becoming the primary medium for programming, communicating and coordinating in the large. Agents are 'wired' to such information infrastructure and their programming and communication practices are molded by it. Interaction, communication and agency become Internet-based and Internet-specific, to the point of becoming unthinkable independently of it. The Internet becomes a generalized medium for software development and knowledge-related work. To it an enormous amount of coordination is delegated, which otherwise would have to be provided by explicit governance mechanisms. Consequently, due to the Internet, creation and distribution of knowledge need very little mediation by market or corporate forms of governance, but directly exploit the connectivity properties of the society. In open-source software environments no theory of knowledge making could ever be developed without incorporating the Internet as a constitutive base of it, as a '*Gestell*' (Ciborra & Hanseth, 1999).

A further instance of technological inscription is represented by software artifacts. Interaction in open-source development projects occurs primarily in a network-mediated computer environment populated with an array of electronic artifacts and tools that Scacchi calls 'software informalisms' (Scacchi, 2001). These web-based artifacts inscribe different kinds of knowledge and help to create a large scale environment for programming and information sharing (Iannacci, 2002: 13). In the following, we only examine source code, mailing lists, and license arrangements of open-source software projects as instances of different types of inscription and as knowledge objects. Source codes (with their multiple versions, interfaces and modules) inscribe technical and

programming knowledge; electronic communication tools and objects such as mailing lists inscribe organizational knowledge, namely knowledge related to the collective practice of developing software; license agreements inscribe legal and contractual rules, being themselves a sort of legal technology. Collective task accomplishment rests on a web of artifacts and tools and on a bundle of technology-based processes that support all knowledge-making practices, to an extent that today most knowledge making and sharing activities would be unfeasible without such electronic gear. Software artifacts and tools populating open-source environments are a legion and are loosely integrated or recombined within and across the ongoing practices and processes. They do not make a fully coherent set of tools, each having its own specific functional destination, but rather resemble a loosely connected collection of available objects that happen to be there in a permanent state of flux, being continuously assembled and discarded. Most of them retain a transient character: they emerge, change or disappear in an ever evolving process of variation and revision. This peculiar mix of traits has led us to introduce the notion of ecology into our picture.

2.3 Ecologies of knowledge

In an effort to capture the evolutionary dynamics of open-source software development a few authors have used the notion of ecology, but none of them has explicitly used it to account for knowledge-related phenomena. For example, Raymond (2001) claims that the open-source community is organized as a business ecology where a variety of interacting agents and available resources keep a dynamic balance. In such ecology no node is strictly indispensible, agents and resources are always replaceable and connected by multiple 'surrogate' channels. With respect to traditional ways of producing and selling closed-source software, the ecology brings two main advantages: the system is more responsive to market demands and is more resilient to shocks. Open-source activity systems have a higher capability to evolve and regenerate themselves because they can always count on a broad variety of participating agents and exploit a large pool of freely circulating resources. In a similar vein, in his study of the LINUX development project Tuomi (2000) points out that open-source software projects are not real 'projects' that implement a pre-

defined plan, but rather an ecology of different development communities, each producing software and hardware products that are then quickly integrated within the overall system as newly available tools and resources (Tuomi, 2000: 11). On their part Healy and Schussman (2003), in a population ecology perspective, use the notion of ecology to describe the remarkably skewed distribution of resources and activities within and between open-source software projects. They find out that only a small number of large projects attract the bulk of development activity and that the internal social organization of large projects is itself very skewed: a small core of highly professional developers build and maintain the code, while around them a poorly differentiated population of lower skill agents are assigned to simpler tasks such as reporting and de-bugging. The authors' findings seems to be consistent with Zipf's classic law of power rank distribution, that characterizes population ecologies (Zipf, 1949).

We use the notion of ecology not in a population dynamics perspective but with reference to the dynamic interaction of multiple heterogeneous elements and relationships, to their competitive or cooperative co-existence, and to the delicate equilibrium that exists between them. In a clear Batesonian vein (Bateson, 1972), for us 'ecology' designates the mix and variety of elements that characterize the activity systems and the practices of open-source development as an evolving domain of practical knowledge and expertise. By using an ecological perspective, we intend to capture a set of opposed but complementary features: variability versus homogeneity, competition versus cooperation, equilibrium versus reproduction, diversity versus standardization, recombination versus ex-ante-design. These dichotomies are part of the dynamics of knowledge making processes in open-source software projects and express the tension between innovation and conservation typical of complex evolutionary systems (March, 1991; Baum & Singh, 1994; Aldrich, 1999). Such 'ecological' character of open-source development gives a special quality to knowledge creation and dissemination. Open-source development activities enact a richly textured knowledge-intensive environment populated with multiple agents and artifacts, tools and practices, crafts and resources, imageries and meanings, all interacting in loosely integrated ways. Projects

exploit the power of diversity to the purpose of knowledge creation. Thus diversity becomes a resource and is turned into a major factor of development and innovation. However, a further major aspect of ecological systems is stability. Open-source software projects are made of heterogeneous components that keep a dynamic balance with one another. The balance does not come from *ex ante* or centrally planned design, but rather emerges out of unplanned, decentral interaction. It is an altogether delicate state, that can be occasionally disrupted by minimal variations.

The idea of ecology applied to knowledge suggests that whatever we call 'knowledge' in open-source software projects is the evolving outcome of the processual interplay of multiple contributions (Bateson, 1972; Sindig-Larsen,1987; Anderson & Laird, 1988). Knowledge comes out of *bricolage*, in which a lot of creative recombination and recycling of existing materials takes place (Lanzara, 1999; Ciborra, 2002). This is consistent with Brown & Duguid's remark that in most domains of professional expertise practices are never completely integrated or highly structured, but tend to be redundant and patched up – a feature that turns into a critical leverage for innovation (Brown & Duguid, 1991). New knowledge hardly emerges in frozen environments but more easily springs out of diversity and surprise, which can only occur in loosely integrated systems where is room for controversies and multiple views.'Ecology' also suggests that knowledge is not a 'thing' that can be purposefully managed, but an evolving 'complex' that can only be fed and cultivated, kept in balance or locally innovated (Bateson, 1972; Hanseth, 1996; Blackler, 1995; Engeström, 1987; Swan & Scarbrough, 2001). No 'knowledge system' is up for grabs as a whole, but can only be peripherally updated.

2.4 Variation, selection, stabilization

In common with all complex systems, knowledge systems need mechanisms for variation, selection, and stabilization in order to evolve. Open-source software projects are a powerful instance of how variation can be effectively combined with stabilization for knowledge making purposes. Variation is the propeller of the process of software development and innovation keeping the development process open to novelties and opportunities. It mainly comes through human agency and is at the

core of all kinds of learning activity in organizations and social systems (March, 1991; Aldrich, 1999). Variation enables software products and software development processes to learn from the environment. It is an unbalancing mechanism that tends to push the system off its path. No knowledge making would be possible without variations.

Building the source code of software is a knowledge-intensive design process. Better or high quality software is a software that has succesfully encoded multiple sources of variation. Projects and designs thus feed on variety. However, too much variation would push variety to such a high level that it could become unmanageable and create excessive instability both in the products and the processes. This in turn would create ambiguity in the market and disarray among the users, who want a good and rich, but nevertheless a stable product. Therefore selection and stabilization mechanims are also needed.

Selection reduces variety by eliminating uninteresting or unconsequential variations and focusing only on the ones that should be kept. Selection always works *ex post,* once that the effects of variation have been experienced and tested. While selection makes variety manageable, at the same time it creates redundancy by producing unexploited opportunities, arrangements and solutions. These are not definitely discarded but set aside to be eventually re-entried into the development process for further re-combinations and re-uses at later times. Selection is a mechanism that splits what is thrown in from what is left out from time to time. Its outcome depends upon value priorities and criteria of relevance, and this is what makes it a knowledge making mechanism. A wide array of selection mechanisms are at work in open-source software projects. They can be explicit, such as authority-based decision rules and voting procedures, or else implicit, such as limited attention resources and technology-based filtering devices.

Stabilization allows for the retention, accumulation and reproduction of succesful experiences. In order to stabilize a selected outcome some repetition and codification are needed (Barley and Tolbert, 1997). Stabilizing mechanisms create memory, standards, rules, patterns of behaviours, structures, and meanings. The emergence of stable system components facilitates and

speeds up system evolution (Simon, 1969). In the following section, we will focus on some specific artifacts and will illustrate how the mechanisms of variation, selection and stabilization of knowledge operate through and upon these artifacts in the open-source environment. The outcomes of our analysis are to be taken as explorative, not conclusive. We try to test the plausibility of our framework and the idea of knowledge ecology by reconstructing some of the most critical aspects of project development.

3. Data and methodology

We have chosen an explorative approach based on the analysis of major open-source software projects. We sampled only two prominent cases (3.1) gathering data from three different sources (3.2). Our data is analysed both qualitatively and quantitatively (3.3).

3.1 Sampling of cases

We selected two cases: the LINUX kernel project and the APACHE HTTP Server Project (see table 3.1-1). LINUX began in 1991 as a private research project. Because of his dissatisfaction with existing operating systems, twenty-one year old Linus Torvalds programmed a kernel resembling UNIX (Torvalds, 2001). The kernel is the part of the operating system that provides its basic functionality. The first version of this program he posted on a newsgroup in the Internet. Following its publication on the Internet, LINUX attracted the attention of thousands of interested users and developers across the world who, in turn, suggested improvements and pointed out errors they had encountered when making use of the system. In the meantime, apart from the kernel project many other sub-projects of LINUX exist.

The APACHE HTTP Server project is a collaborative software development effort aimed at creating a robust, commercial-grade, and freely-available source code implementation of an HTTP web server. The basis of APACHE was the public domain HTTP daemon developed by Rob McCool at the National Center for Supercomputing Applications (NCSA) at the University of Illinois. When McCool left NCSA 1994, a small group of web masters who had adopted NCSA

server software for their own web sites decided to continue the development for themselves. They contacted via private e-mail, gathered together for the purpose of coordinating their changes (in the form of 'patches'). The name of this 'patchy' web server software evolved into APACHE. After extensive feedback and modification by users, APACHE 1.0 was released on December 1, 1995. In 1999, members of the APACHE Group formed the APACHE Software Foundation to provide legal and financial support for the APACHE HTTP Server.

Table 3.1-1: LINUX and APACHE

The projects were selected for two reasons: firstly, they are far developed and in a mature and stable

	Aim	Starting	Start	Actual Phase
LINUX	Stable operating system based on UNIX-architecture	Development based on MINIX from the Finnish student Linus Torvalds	1991	• Optimized, stable version (2.4.20) • Adaptions to new hardware
APACHE	Robust HTTP-server for modern operating systems	Continuation of the development of the HTTP DAEMON server as a „patchwork" with eight webmasters	1995	• Optimized, stable version (2,0) • Permanent development

phase of development. Secondly, a huge number of participants is involved in both projects, what makes issues of knowledge creation and coordination more interesting than in smaller projects. Focusing on only two cases has the advantage of increasing the depth of the analysis. As we could see from our data, the amount of activity and participation at LINUX is comparatively higher than at APACHE. Furthermore, LINUX is still highly characterized by its founder Linus Torvalds, whereas the APACHE HTTP project is together with other open-source software projects integrated under the Apache Software Foundation. Although we were interested in detecting some variance, we did not quite take them as two distinctive instances of how knowledge is made and shared in

open-source software projects. In spite of the different project goals, size and contents, it turned out instead that the similarities in process, structure and behavior are more conspicuous than the differences.

3.2 Data gathering

For data gathering, we used three different sources. Firstly, in order to obtain a rudimentary understanding of the projects we analysed the *projects' web pages*. We examined the project's history, its structure, aims, core activities, and the legal framework in form of license arrangements. These data sources were critical for understanding the mechanisms of variation, selection, and stabilization by tracking the process of 'versioning' and the production of updates of the software.

Secondly, we conducted twelve *personal interviews* in two rounds with four LINUX developers and two APACHE developers. The interviews of the first round were explorative and unstructured. Each interview took between one and two hours. In order to obtain an in-depth understanding of the open-source software phenomen we tried not to lead any of our interviewees' responses, and not to share our knowledge or our guesses with them (Miles & Huberman, 1984). The interviews of the second round were semi-structured with guidelines including questions concerning the processes of knowledge-making in open-source software projects. In part, the interviews helped us to check and eventually correct the picture that we were developing through our navigations and observations across the web.

Our third source of data gathering were the *project's e-mail conversations* archived in the projects' mailing lists. Focusing on the development of the software, we gathered our data from the developer mailing lists lists. The projects have also other mailing lists for bug reports, user support and announcements of code changes. But the general discussion of development topics (for example technical details, errors, projects design, emerging architecture, and announcements) takes place at the developer lists. Anyone with an interest in working on APACHE or LINUX can join the respective developer mailing list. Both lists are archived monthly under a publicly open mailing list.

3.3 Data analysis

The projects' websites and the interviews are analyzed qualitatively. Whereas the e-mail analysis forms the basis for the quantitative analysis. The units of the e-mail-analysis are communication sequences of the developers in the form of e-mail-threads. A thread is a sequence of e-mail messages around a common discussion theme with a distinct heading, for example a question or the posting of a new patch. Each first mail of a thread is expected to generate conversational activity that potentially extends into the following days, weeks, perhaps even months. Agents produce the thread by asking questions, giving answers or generating other communication activities via e-mail. Using open coding (Strauss & Corbin, 1990), we developed a coding scheme for the content of the first e-mail of a thread. This provided eleven different first mail message types, including such items as reporting errors of the software ('bugs'), providing new parts of the software ('patches'), asking questions concerning the use of the software, and announcing new versions or new patches. We created a database of all threads including identities of the thread initiatiors, date and time for posting the different e-mails, and content of the first mails for the period November, 15th – November, 30th, 2002: a casually chosen fortnight in the life of the analyzed projects. In choosing a period at random, we assumed that, as the mailing lists reflect mundane everyday conversational and crafting activities for the programmers, what happens in a relatively short time period chosen at random in the project's history can be taken as a plausible example of the general pattern of behavior and knowledge making. In other words, the pattern of ongoing interactions should tell us something about the dynamics of knowledge making and sharing in the project. In order to test this assumption we also performed the data analysis for one single day (november 30th) and we found similar patterns. The complete database included 4600 single e-mail contacts in 1256 threads for LINUX initiated by 573 participants, in APACHE 465 e-mails in 88 threads initiated by 44 participants (see table 3.3-1).

	Number of threads	*Number of mails*	*Number of participants*
LINUX	*1256*	*4600*	*573*
APACHE	*88*	*465*	*44*

Table 3.3-1: Basic data for APACHE and LINUX (November, 15th-30th, 2002)

4. Knowledge making and sharing in open-source software projects: artifacts as inscriptions and carriers of knowledge

We have used the different data in order to examine three kinds of artifacts, respectively inscribing technical, organizational, and institutional knowledge. For 'technical' we analyze how new versions of the source code are generated, updated, and stabilized (4.1); for 'organizational' we analyze the structure of discussion threads found in the project's mailing lists (4.2); for 'institutional' we analyze how license agreements induce and protect institutional environments that enhance both stability and innovation (4.3). For each of them we describe the different activities and the network dynamics in which they are embedded (as an overview see table 4-1).

We show how the different artifacts are affected by the mechanisms of variation, selection, and stabilization, and in turn support such mechanisms. We regard artifacts not as bounded 'repositories' where knowledge is held and stored, but rather as active inscriptions of practical knowledge that in turn support all knowledge making and sharing. As we see them, artifacts are 'dynamic vehicles' that make knowledge dissemination possible by circulating throughout the Internet. They all interact in the Internet-based knowledge environment, becoming primary tools for coordination.

Mechanisms / Artifacts	Variation	Selection	Stabilization
Source Code	• Openness • Frequent releases ('versioning') • Modular design • Recombination	• Rules • Selective decisions • Functioning	• Standardized interfaces • Documentation • Parallel versions & modules
Mailing Lists	• Free accessibility • Multiple agents • Quantity of threads • New problems • New solutions • Knowledge transfer across lists	• Selective participation • Few mails per thread • Short lifetime of threads	• Repetitive participation • Documentation • Nested Communication
License Arrangements	• Guarantee of free access to knowledge • Co-presence of different license arrangements	• Channeling of behavior • Consolidating norms of behaviour	• Institutional environment • Assumptions about knowledge • Practices of open-source • Rules of reciprocity

Table 4-1:   *Artifacts and knowledge processes in open-source software projects (overview)*

4.1 The source code as a knowledge organizer

The source code is the set of instructions that developers write when creating programs. It inscribes the basic programming rules that run the software, but it also turns into an organizer of programming activity and knowledge. It is affected by the mechanisms of variation, selection, and stabilization. At first, the source code is a playground for *variation* keeping the process open to a wide range of novel knowledge. Our data reveal four critical aspects influencing the variety of the code: openness, frequent releases ("versioning"), modular design of the software, and recombination. The *openness* of the source-code creates the basis for its variation into new software versions. It is guaranteed by different license arrangements (see 4.3). The users can modify the source code according to their own purposes. They can use the new software or even redistribute it under a new name. Out of this bifurcation new programs emerge and eventually evolve into new bifurcations. Sometimes they even compete with the original software ('forking').

The *frequent releases* of new software versions maintain and even enhance the variety of knowledge in open-source software development. New versions are released more often and earlier

than in usual proprietary software development (Raymond, 2001). The versions are arranged in different series. For example, at LINUX in the last nine years (1994-2003) nine new series were released (see figure 4.1-1).
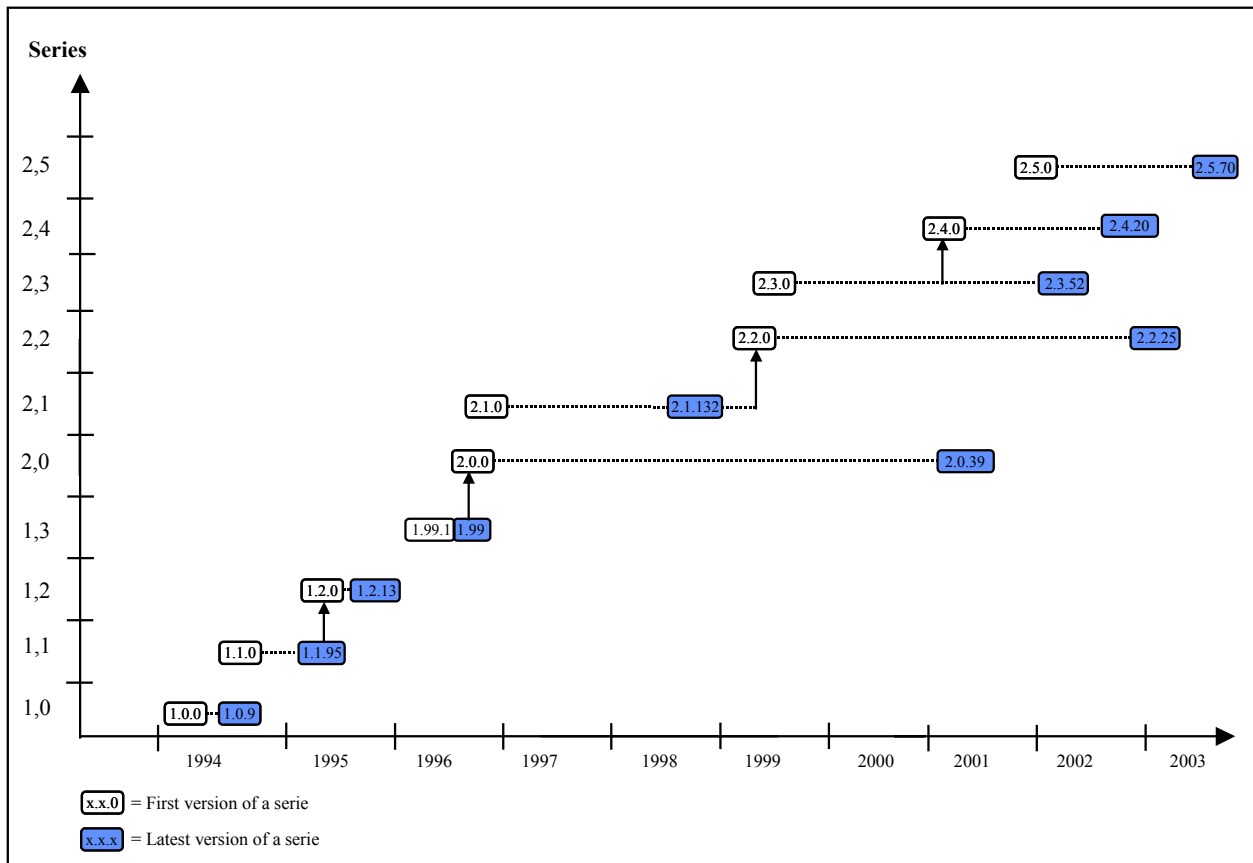

Figure 4.1-1: Latest LINUX kernel versions of different series
(Source: http://www.linuxhq/kernel/index.html)

Every serie has different versions beginning with version x.x.0 and ending with the latest version of that serie. Inbetween are as many versions released as the last number of the latest version shows. For example, between version 2.5.0 and 2.5.70 lie 69 versions. Fundamentally, versions are divided in two series: production (even series, e.g. 1.2.xx, 2.0.xx or 2.2.x) and experimental (odd series, e.g. 1.3.xx or 2.1.x). Production or stable series have a well defined feature set, a low number of known bugs, and tried and proven drivers. So from version 2.0.2 to 2.0.3, there were only bug fixes, and no new features. From time to time, as the current development series stabilizes, it will be frozen as the new 'stable' version, and development continues on a new development version of the kernel. The experimental series are fast moving versions that are used to test new features, algorithms, device

drivers, etc. By their own nature the versions of experimental series may behave in unpredictable ways, so one may experience data loss, random machine lockups, etc. They are released much more frequently than the production versions. Furthermore, intermediate pre-release versions exist.

Responsible for the different versions at LINUX are different persons called 'maintainers'. Linus Torvalds has delegated maintenance for older stable versions to others, while he continues development of the latest 'bleeding edge' release. The maintainers at LINUX have different attitudes towards new drivers, or new patches in their respective version. For example, while Linus Torvalds is known to be very conservative and only applies obvious and well tested patches to the 2.4 kernel, Alan Cox maintains a set of kernel patches that contains new concepts, more and/or newer drivers, and more intrusive patches. If the patches prove themselves stable, Alan submits them to Linus to include them into the official kernel. This enlarges the variety of knowledge of the source code.

What makes the parallel decentralised processes of open-source software project development and therefore the variation of the code feasible is the *modular design* of the source code. Only piecemeal built software allows the developers to work independently on the various components (Moody, 2001: 14 and 82). Modularity implies recombination and flexibility: modules can be changed, moved, and substituted independently from each other − thanks to standard interfaces. The source code of APACHE for example is built extremely modular. It is made up of a small core with basic server features and contains 50 further software-modules. Six different modules exist already for multi-processing. And as well LINUX, known for its monolithic kernel, consists of different modules according to its UNIX-like modular structure. Apart from the kernel, 82 LINUX software projects are announced at the LINUX website. This kind of modularity allows changes to be implemented module-wise without any risk of having a negative impact on other parts of the software. Furthermore, there are many extra patches to the kernel for new features. Most of the 'new' solutions and patches originate from *recombination* and re-use of fragments and materials that are available on the web, being the product of previous programming work.

Selection of the source code reduces variety in order to focus on the variations that are to be kept. It makes variety manageable. The variety of source code is harnessed by a large set of different rules concerning programming and de-bugging, by selective decisions of core developers, and by the functioning of the software. *Rules* structure the systemic memory and determine for a system what is to be retained and what can be discarded (Luhmann, 2000). For example at APACHE exists an official style-guide that shows the developer how to write patches and send in suggestions for improvements or pointing out errors. LINUX announces thirteen official Kernel-Hackerguides at its web-site, that should be read by the prospective developer before contributing. Furthermore, so-called bug-reporting formats exactly prescribe the way errors should be reported in order to be considered. Other rules can be read under the Frequently Asked Question-Website (see table 4.1-2).

Are there any implicit rules on this list that I should be aware of?

- Stick to the subject. This is a Linux kernel list, mainly for developers.
- Use English only!
- Don't post in HTML format! If you are using IE or Netscape, pleas turn off HTML formatting for your posts to the kernel list.
- If you will be asking a question, before you post to the list, try to find the answer in the available documentation or in the list archives. Just remember that 99% of the questions on this list have already been answered at least once. Usually the first answer is the most detailed, so the archives contain far better information than you will get from somebody who has answered the same question a dozen times or more.
- Be precise, clear and conise, whether asking a question or making a comment or announcing a bug, posting a patch or whatever. Post facts, avoid opinions.
- Be nice, there is no need to be rude. Avoid expressions that may be interpreted as aggressive towards other list participants, even if the subject being treated is particularly relevant to you and/or your controversial.
- Don't drag on with controversies. Don't try to have the last word. You will eventually have the last word, but meanwhile you'll have lost all your sympathy credit.
- A line of code is worth a thousand words. If you think of a new feature, implement it first, then post to the list for comments.
- …

Table 4-1-2: Rules on the LINUX-kernel mailing list FAQ (Source: http://www.kernel.org/pub/ LINUX/docs/lkml/)

The final *selective decision* of how to change the source code of the original program is reserved to a very small number of persons compared to the number of participants. In the case of APACHE for example 22 developers of the core development team decided (during the period from February

1995 until May 1999) on 3975 bug reports reported by 3060 users and on 6092 code submissions as well as 695 bug fixes handed in by altogether 432 developers (Mockus et al., 2002). In the case of LINUX the last decision concerning changes in the code of the latest version of the kernel belongs to Linus Torvalds. Whereas the latter decides as a 'benevolent dictator' without any given rule, at APACHE exist exact voting rules that are codified in the APACHE-Project-Guidelines. In both cases, selection takes the form of deciding on the relevance of knowledge and therefore its integration in the source code. Obviously, here the interactive system is complemented by organizational mechanisms like decision rules and authority.

A further important mechanism in the selection of source code is made by the *functioning of the software*. If the patches do not work, there exists no possibility of integration in the final version. And because there are many smart users who are available and willing to test the code, and ready to share the results of their trials with their peers, the testing is manifold and severe. This leads us to remark that a critical and very effective mechanism for generating knowledge (or selecting it out) is built into the very structure of the code, which is always 'exposed'. Openness, versioning, and modularity with its recombinant properties make the code always available for inspection. The code 'calls' for being repeatedly checked out. Developers do respond to the call and keep producing patches and new versions that are fed back into the system to generate further problem-solving work and further innovation.

Along the development process the source code *stabilizes* technical knowledge thus allowing for the retention, the accumulation, and the sharing of it. In our case studies we found three main stabilizing devices: standardized interfaces connecting the different modules, documentation of the code, and the functional equivalence of parallel versions and modules. Modularity not only produces variety, but also allows stabilization. But in order to reach stability, the modules must be on the one hand independent of each other, and on the other hand easily to integrate. This is resolved by *standardized interfaces* between the different modules of the code. In

this way at LINUX exist different interfaces for different drivers which can be easily loaded into the kernel.

An important part of stabilizing technical knowledge in the form of the source code lies in its *documentation*. The documentation as a part of the systemic memory enables a selective re-utilization of the irretrievable past (Luhmann, 2000). Documentation allows communication to be reproducible, and therefore increases the probability of being followed by other communications. In open-source software projects the source code is usually documented in version-control systems such as CVS ('Concurrent Version System') or Bitkeeper.[1] They synchronize work and keep track of changes in the source code performed by developers working on the same set of files. At LINUX as well as at APACHE mainly CVS is used. CVS is a public version control tool that stores its version-control information in a directory hierarchy on a central server. This allows developers to add or remove files easily, or to ask for versioning information of files. Each change of the code is noted with some identification of the person making the change and this identification, together with the date, time and size of change is automatically recorded. CVS allows anybody to check out code from the repository. However, changing the source code is restricted to few developers.

Furthermore, redundancy of *parallel modules* and *versions* creates stabilization because of their functional equivalence. Different modules and versions exist in parallel, partly with same functions. For example, at LINUX there does not only exist a latest beta-version (2.5.69) and a latest stable version (2.4.20), but as well some former stable versions are still maintained (see again figure 4.1-1). The versions are stabilized by so-called feature and code freezes. A feature freeze at LINUX for example is when Linus Torvalds announces on the LINUX-kernel developer list that he will not consider any more features until the release of a new stable kernel version. A code freeze is more restrictive. It means only severe bug fixes are accepted. This is a short phase that usually precedes the creation of a new serie.

---

[1]   CVS: http://www.cvshome.org; Bitkeeper: http://www.bitkeeper.com.

The striking feature in the process, that we have briefly described, is that the source code of the software becomes an organizer of human knowledge and agency. More than a frozen repository of established knowledge, it is a tool for knowing and organizing. The source code is a 'warm' object. Programmers attend to the implementation of the code and the code in turn 'talks back' to the programmers through its multiple beta versions, bugs and ever evolving features, thus orienting the programmers' future contributions through the functionalities and, perhaps more critically, the dysfunctions (bugs) being inscribed. Beta versions that are developed serially are never finished products, but rather 'for instances'. Users can 'play' with them in order to test them and see if they work. Through the 'for instances' the programmers establish cognitive transactions with their partners and with their building materials, stay for them for a while, and then go ahead for the next moves. The code plays a central role in these cognitive transactions. Every new version or patch that is posted on the Internet triggers new cycles of activities and communications, and as a consequence also new bugs and misunderstandings are produced, which in turn give rise to a new round of activities. The participants communicate through the code by changing it - always referring to certain versions, patches, features, and bugs. In this way, the code becomes a focal point, playing a major signalling and coordinating function (Schelling, 1960). Accordingly, the status of the technical knowledge inscribed in the source code and in the software is very peculiar. It is transient knowledge: it reflects what has been programmed and developed up to that point, resuming past development and knowledge and pointing to future experiments and future knowledge.

4.2 Discussion threads: Sharing knowledge at electronic crossroads

Outcomes and problems of the programming work are discussed and refined in an ongoing stream of e-mail conversations. Mailing lists are virtual work environments where the communications among multiple agents take place (Lee & Cole, 2000). They fulfill three different functions concerning the creation and circulation of project knowledge. First, they are virtual construction sites where software constructs are continuously created and updated, modified and repaired (that

is, places where people do the actual programming work). Secondly, they are a sort of electronic crossroads where information is exchanged and problems and solutions are discussed (that is, places where people talk about the work they do); thirdly, they are web logs where the history of open-source software projects is recorded (that is, places where the work done and the talking are documented). Thus, an analysis of such artifacts is critical in order to understand the knowledge dynamics of an open-source software project, because it reveals both the time structure of the conversations that an undefined set of programmers entertain around specific coding issues and the knowledge content of the communications. On mailing lists we find communications, enquiries, commentaries, peer reviews and evaluations, bug reports and answers to questions, nested conversations leading to problem (re)definitions, asynchronous negotiations and ongoing controversies. All these activities are continually carried out, across the world and around the clock, and play a pivotal role in creating both the software and the knowledge about the software (Iannacci, 2002). At the same time they give us information about the structure of social interaction supported by the communication technology. These artifacts are dynamic, ever changing repositories and carriers of problem solving knowledge within and across projects and sub-projects. Through them, knowledge is circulated all over the projects and the programming environment. This is why we like to think of them as 'crossroads' where people meet, exchange information, and attend to common tasks.

In order to describe the interplay of variation, selection, and stabilization, we analyzed the developer mailing lists of APACHE and LINUX during the period of November 15th to 30th, 2002. Units of analysis are communication sequences in form of e-mail-threads. They tell us about the structure and the dynamics of knowledge making and sharing in open-source software projects. In the text, we highlight some of the most relevant features of our so far preliminary findings. Although the analyzed projects differ in terms of activity and participation, they show similar patterns with respect to the mechanisms of variation, selection, and stabilization.

In both projects, although in different degrees, we notice a high production of *variety*, due to the free accessibility, multiple agents, high number of started threads including a lot of mails with new problems and solutions, as well as knowledge transfer across different lists (see again table 4-1). Fundamentally, the analyzed developer mailing lists are *accessible by anybody*. Everybody who wants to can register himself and post his comments on mailing lists. Day after day, *multiple agents* exchange messages and generate activity around multiple issues concerning the software. Whatever the message is, it will be posted on the list or board and circulated across the network, thus becoming potentially accessible by a variety of agents. Responsiveness is encoded into the network and the communication technology: agents see that their input matters, that they get answers to their questions, and are willing to participate into the process of continual updating and innovation (Neff & Stark, 2002:6). By entering the conversations around specific issues users and developers generate a thread. Threads are given life and structure by time, reciprocity, and attention. The *quantity of threads* generated in the analyzed projects builds the basis of variety. In the chosen period, at APACHE 88 threads with altogether 465 e-mails were initiated by 44 participants. At LINUX even more 'mass' is generated: 1256 threads including altogether 4600 mails were initiated by 573 participants. At LINUX an average of 78,5 threads is created per day with a range of activity from 46 to 125, while in APACHE the average is 5,5 threads per day with a range from 0 to 17 threads.

Concerning the content of the communication, variation is produced by *new solutions* on the one hand and *new problems* on the other hand. We coded the first e-mail of a thread according to its content (see section 3.3 and figure 4.2-1). Problems are questions and problems in using the software (Q), the identification of errors – called 'bugs' (B), and general questions concerning further modules, versions, etc. (GQ). Solutions are new patches that are provided for further development of the software (P), suggestions for improvement of certain aspects (SI), and comments or answers to former threads (C). The data show that most of the communication in the developer mailing lists concerns new patches (P) that are provided for further development of the

software (39,17% for LINUX, 19,33% for APACHE) or questions about problems using and developing the software (25,80% for LINUX, 39,77% for APACHE). It is interesting to notice that at APACHE more threads (43,18%) begin with new problems (Q, B, GQ) than with solutions (32,95%) offered to different problems (SI, C, P), whereas at LINUX offering solutions (47,45%) predominates the reporting of problems (34,63%) in the first mails of the respective thread.
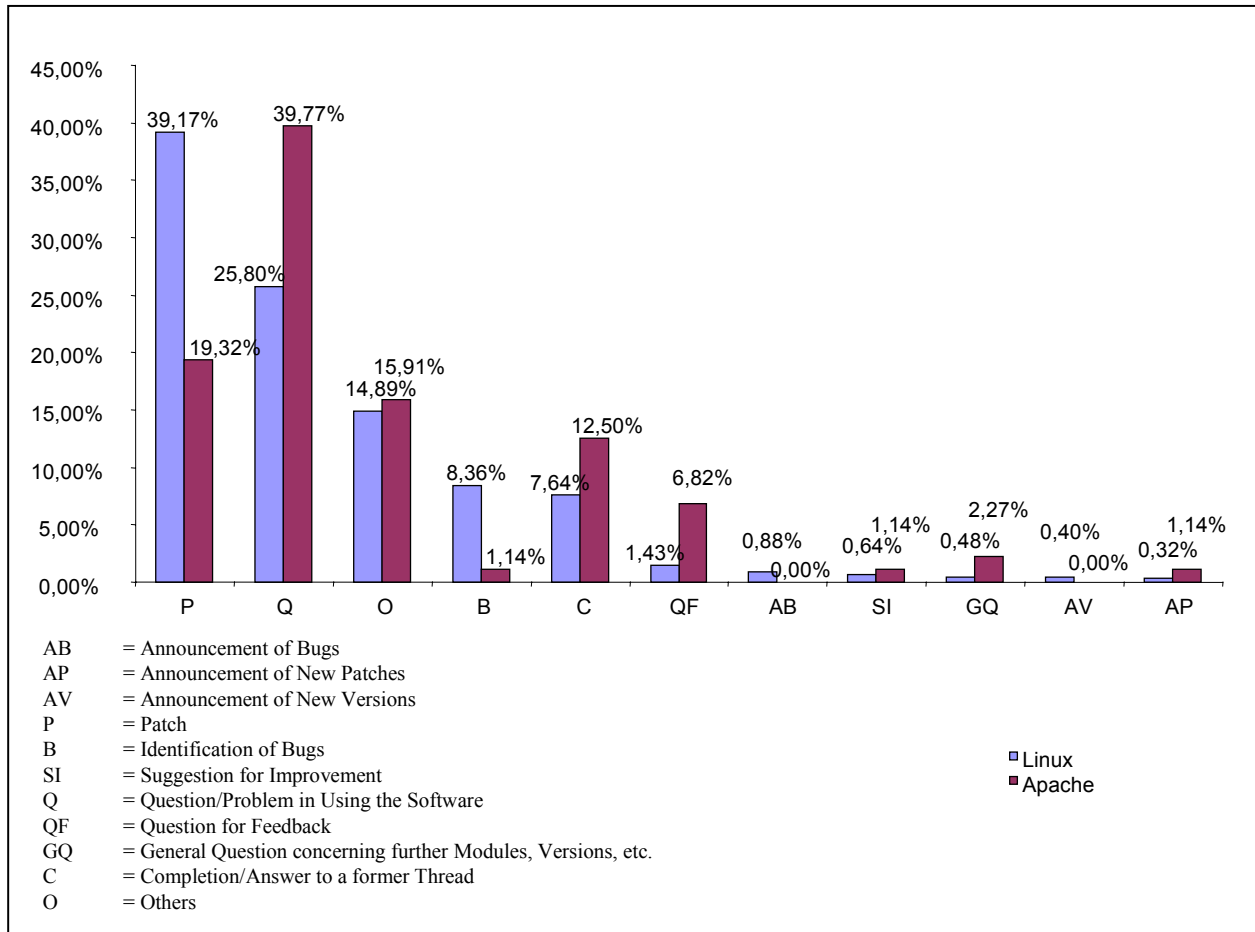

AB	= Announcement of Bugs
AP	= Announcement of New Patches
AV	= Announcement of New Versions
P	= Patch
B	= Identification of Bugs
SI	= Suggestion for Improvement
Q	= Question/Problem in Using the Software
QF	= Question for Feedback
GQ	= General Question concerning further Modules, Versions, etc.
C	= Completion/Answer to a former Thread
O	= Others

Figure 4.2-1: Content of first mails per thread for APACHE and LINUX (Nov., 15th-30th, 2002)

Interestingly enough, we found evidence of *knowledge transfer across different lists* of different projects. In some cases the discussion took place completely parallel by posting all mails concurrently at different lists. That shows that projects are not closed, self-referential worlds. Rather the communication is 'infected' by other lists.

Variation is followed by *selection*. A large amount of information comes in, but only a little finds its way through the system and gets processed. Threads emerge selectively from generalized interaction, signalling some purposeful activity. Most of them are intermittent, ephemeral, volatile

objects. Just minimal coalescent structures, fed by interaction (Weick 1993). We identified three aspects of selection in analyzing the mailing lists: selective participation, few mails per thread, and short lifetime of threads. Even if the overall interaction is potentially all-to-all and every developer enjoys potentially free access to all information on the web, the actual *participation* is very selective. Very few threads have more than a handful of participants. In APACHE we found that only 23% of the analysed threads have more than three participants, at LINUX only 14% (see figure 4.2-2).
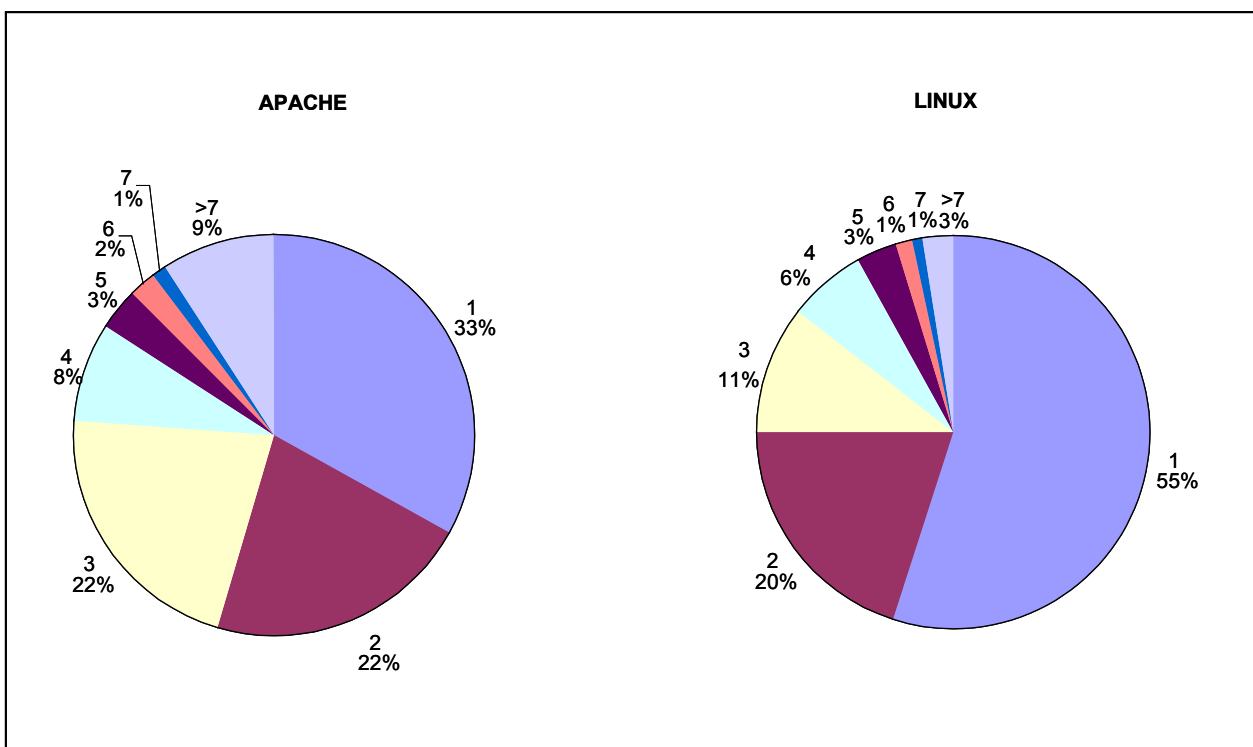

Figure 4.2-2: Participants per thread (in %) for APACHE and LINUX (November, 15th-30th, 2002)

The average number of participants per thread is very low (2,2 for LINUX and 3,1 for APACHE). The average number of *mails per thread* is low, too (3,7 for LINUX and 5,2 for APACHE), with a minimum of one for both projects (messages without answers), but with remarkably high maxima (see table 4.2-3).

	Average number of participants per thread	Average number of mails per thread	Average number of days per thread
LINUX	*2,2* *(max=36; min=1)*	*3,7* *(max=95; min=1)*	*2,9* *(max=135; min=1)*
APACHE	*3,1* *(max=12; min=1)*	*5,2* *(max=40; min=1)*	*9,7* *(max=174; min=1)*

Table 4.2-3: Averages for APACHE and LINUX (November, 15th-30th, 2002)

At LINUX as many as 75% of the analysed threads have less than four mails per thread and only 9% of threads have more than eight mails. In APACHE the pattern is the same, although the figures are a bit different: 50% of the threads have less than four e-mails per thread and only 15% reach more than 8 mails per thread (see figure 4.2-4).
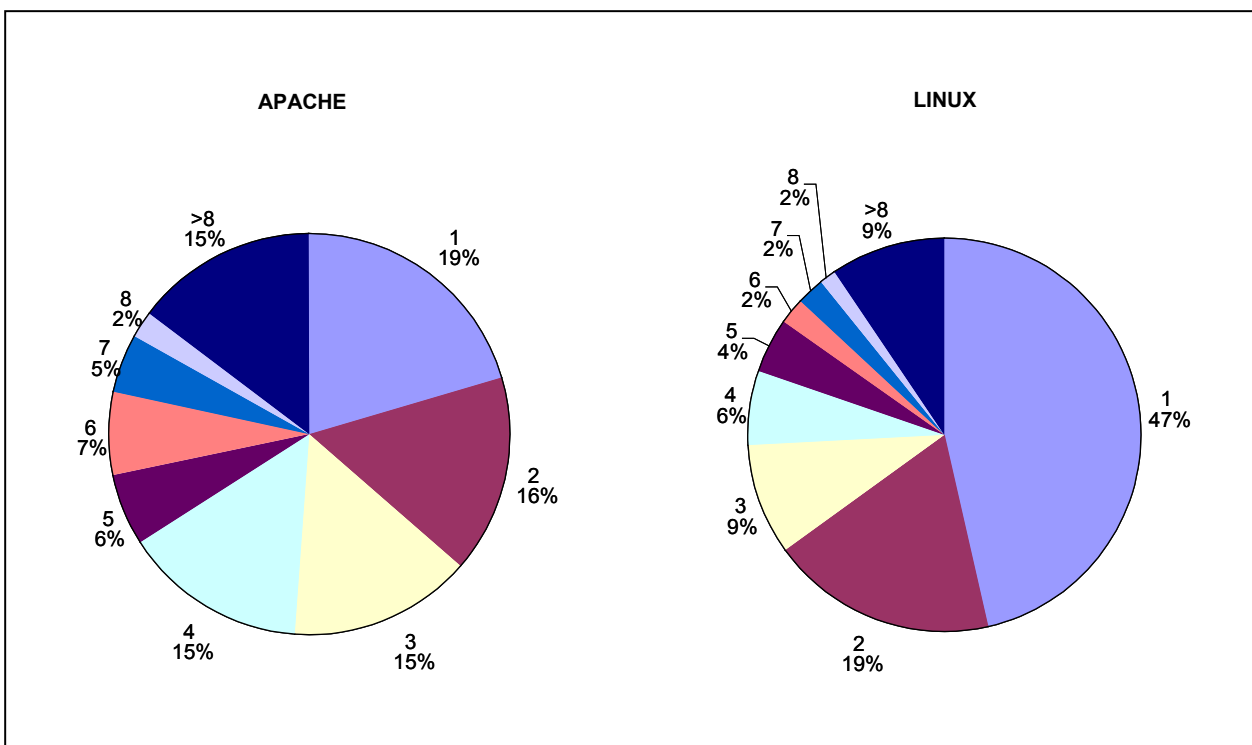

Figure 4-2-4: Number of mails per thread for APACHE and LINUX (November, 15th-30th, 2002)

If we track the number of mails per thread, we notice that, especially at LINUX, a high percentage of first mails are left unanswered and do not have a follow up (LINUX 47%; APACHE 19%). In

this cases, the thread is made of one participant and one mail, so basically there is no thread at all. Considering the content of the first mail, at LINUX even 38,75% of direct questions are left unanswered (17,14% at APACHE).

The average *lifetime* of a thread is generally very short: 2,9 days in LINUX, 9,7 days in APACHE, with a minimum of one day for both projects and maxima over one hundred (see table 4.2-3 again). Most of e-mail conversations never really come to form a stable and lasting thread. In LINUX only 30% and in APACHE 55% of threads reach an age of more than one day (see figure 4.2-5). Taken individually, threads tend to be short-lived, ephemeral structures, but as a whole they make persistent bundles or streams of variable either thickness or thinness.
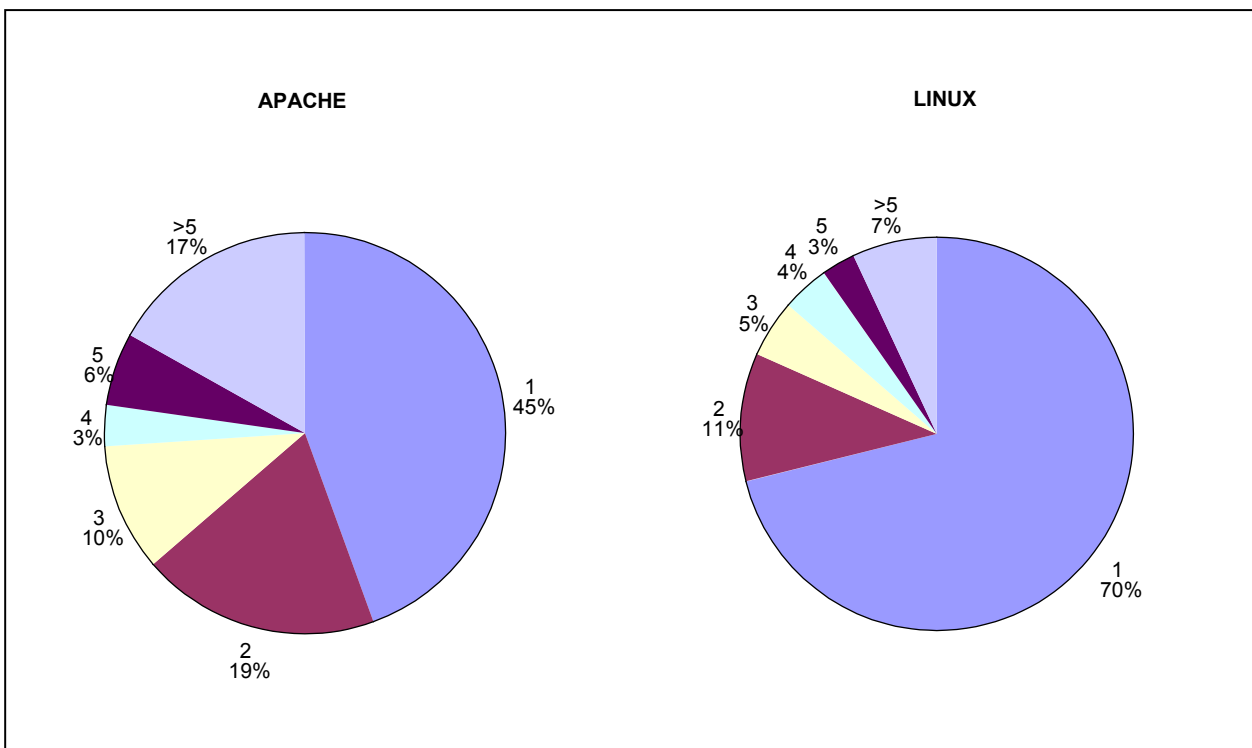

Figure 4.2-5: Days per thread (in %) for APACHE and LINUX (November, 15th-30th, 2002)

Selection of variety means as well dissipation. We found a great amount of dissipation in the activity system. Basically, a lot of effort is 'eaten up' by the system. It becomes thrash. The reasons for this can be many. One is simply that there are not enough attention resources in the system to process everything that comes in the mail. Indeed, although we found some exceptions, e-mails that are not answered the same or next day are seldom answered at all. One does not go back to them, because there are others coming onto the screen, and they just fade off. As a second reason, it could

also be a signal that only a few questions and problems are perceived as worthwhile to be pursued by the developers and singled out of the many to be worked upon. Another path of explanation could be based on the technology. The e-mail technology inscribes informal rules for communication that are quite directive: 'you respond immediately to what most concerns you and to what is immediately relevant for your ongoing daily work, otherwise you respond later or do not respond'; 'you do not respond at all to what is out of your range of interest'. Dissipation in open-source software projects is not necessarily a source of inefficiency, it is a source of redundancy that can be usefully exploited in the overall system dynamics: a distributed reserve of low cost knowledge and attention resources is always available to feed the project development needs. Specific patches capture the attention of a few and discussion starts. Activity clusters around a few issues or themes at a time, on which attention is selectively focused, generating addensation of agents. Of course, there is always the possibility that an important signal, warning, question, or original solution is neglected, or that people pursue tracks that lead nowhere, but the high number of participants that provide simultaneous scrutiny minimizes the chance, overall. In this connection, it is interesting to highlight the different figures for variation and selection in LINUX and APACHE. The very same mechanisms are at work in both project, but in APACHE there seems to be less dissipation than in LINUX. On average in APACHE more participants discuss with more intensity (no. of mails) for longer periods. This leads us to think that in APACHE connectivity is tighter and communication is more stable in time. This is due perhaps to the different nature of the task, or to smaller number of participants, or to higher professionalism of the core developers. Indeed, the very high participation and dissipation rates in LINUX activity are an indicator of LINUX broader popularity. These features may have interesting implications for the understanding of knowledge making and sharing that need to be further explored.

Only in a limited number of cases the communication is *stabilized* and coalesces into conspicuous and relatively persistent threads. The longest threads are 135 days at LINUX, and 174 days at APACHE (see table 4.2-3 again). In the mailing lists we found mainly three aspects

stabilizing the communciation: repetitive participation of same participants, documentation and nested conversation. Usually, in longer threads the same agents *participate repetitively*. For example at LINUX, the threads with five mails are written from 3,4 participants on average. Threads with 19 mails are written from only 8,0 developer on average. At APACHE at threads with five mails 4,0 agents participate on average. Just the same way as for the source code (see as well section 4.1), the communication via mailing lists is stabilized by its quasi-automatic *documentation*. The mails are archived and everybody has access to them. One special way to use the documentary quality of written communication via e-mails is the mean of *nested conversations*. Typically, the parts of a previous message to which the following message refers are inserted into the new one. Iannacci (2002) labels such interactions as 'dialogic negotiations' which imply a continuity and interdependence among messages as indicated by the inclusion of previous messages or pieces of messages, thus creating 'metaphorical conversations'.

Robust and durable threads emerge very rarely but the flow of communication is never completely discontinued. The single threads go off and on, but the overall fabric of knowledge making and sharing, though it might become thinner or thicker, never collapses. At any point in time in the development of the project there is always a bundle of active threads that signal and carry on activity. Thus the continuity and the stability of the project are assured by the ongoing emergence and disappearance of threads along a main stream of sense- and knowledge-making.

4.3 The legal rules: Enacting an open environment for knowledge making and sharing

The openness of the source code is protected by license arrangements. Open-source licenses have four things in common: the right to earn license fees is typically waived, further distribution is allowed, no usage restriction exist, and the condition that the source code is made available is incorporated (Spiller & Wichmann, 2002). The major distinction of different licenses is the principle of 'copyleft'. It was created within the scope of one of the first and the most widely used license arrangement: the General Public License (GPL) that LINUX and many other projects

operate under to this day.[2] The GPL was developed by Richard Stallman and aims to prevent the commercialisation of cooperatively developed software. Users and developers are not allowed to place their own license restrictions on the refined software (Stallman, 1998). Instead derivative works have to be released under the same license again. Stallman terms this 'copyleft' - a play on the word 'copyright'. Over the years, further license agreements have been drawn up, in turn forming the legal background to the evolution of open-source software (McGowan, 2001; Moglen, 1999). For example, contrary to LINUX which belongs to the GPL, APACHE is non-copyleft software.[3] The APACHE Software License (ASL) does not claim that the derived work must be free again. Furthermore, the developed software can be linked with proprietary software, thus enacting a sort of hybrid regime.

The license arrangements are legal artifacts that result from the inscription and translation in legal form of rules of conduct which are already implicit and embodied in the Internet technology infrastructure. They give a legal clothing to the actual practices of the software developers and programmers on the web, being fully compatible with an open knowledge environment where information circulates freely and there is broad information feedback and re-entry. Basically, with the copy-left rule the habit of programmers and hackers of posting their piece of code on the web and circulating it to other programmers so that it can be improved through peer review takes a legal form. Therefore, collectively creating and sharing knowledge is a practice, which is subsequently given an institutionalized form. For example, all licensing agreements take for granted that development of innovation is user-based, and that users' dispersed knowledge generates software of higher quality than strictly corporate-based knowledge. They imply that users are designers and 'customizers' of the software, not simply buyers of a finished product.

As a *variation* mechanism, license arrangements facilitate entries and participation of new programmers and users into the development process. The new agents bring their own skills and needs with them, hence new knowledge, which turns into a constant feed for the system. That

---

[2]   See http://www.gnu.org/copyleft/gpl.html.

enormously enriches the pool of knowledge resources the open-source activity system can rely upon at any given stage of development. The open-source license agreements, and in particular the copyleft rule, enact a sort of 'obverse' legal and contractual regime where *access to knowledge* is not restricted or ruled by the market. A legal regime of this kind has positive network externalities, that result in constant pressures toward experimentation and innovation: if anybody in principle is allowed to modify the code and design software, then software becomes more and more an evolving artifact, and the knowledge that it embodies and carries around is created through an ongoing evolutionary process.

Together with opportunities the license arrangements create also barriers for knowledge creation and dissemination. For example, the very strong 'copyleft' makes the GPL not very business-friendly because any software company would have to reveal their software source code if they included parts of GPL software to develop it. From this point of view, the *co-presence of different legal and contractual regimes* generates flexibility and facilitates innovation. For example, the possibility to do linkages with proprietary software that exists for APACHE, but as well for Perl (Artistic License) and Mozilla (MPL), enhances the variety of the development. The software can be incorporated into software product that can be licensed without 'contaminating' the new software.

As a *selection* mechanism, the license arrangements *channel individual and collective behaviours*. They formally consolidate the *norms of reciprocity* and mutual openness that belong to the programmers' shared culture. However, different kinds of licenses institutionalize different sets of reciprocity norms: in either one, there are different things and transactions that can/cannot be done with the software (see GPL and APL). Especially, the 'copy-left' of the GPL curbs private appropriation and free riding.

As a *stabilizing* factor, copy-left induces and stabilizes an *institutional environment* for programming in the large where it makes sense to do collaborative development work, have open conversations and unbounded communications. It embodies *assumptions about knowledge* as a

---

3   See http://www.apache.org/foundation/Licence.

public good, its generalized accessibility and usability, the value of geographically dispersed skills, and finally about how knowledge should be created and used in an open society and in a user-driven innovation process. Basically, copy-left is a set of rules that regulates the distribution of expert knowledge and the access to it. As all sets of established rules, it tends to stabilize behaviours and dispositions that are specific to that rule-set, namely the actors' willingness to reciprocate (Iannacci, 2003:21). Also, specific habits and cognitive frames are enhanced, so that we may speak of the *practice of doing open-source*, or the habit of being in an open-source programming environment, or having an open-source mindset.

In synthesis, while copy-left and other components of open-source software licenses might seem revolutionary arrangements, that turn the standard ways of producing and distributing software upside down, at the same time they consolidate generalized *rules of reciprocity*, thus enhancing generalized access to knowledge and collective problem-solving capabilities. Copy-left is nothing else but the final legal inscription of a new way of creating and distributing knowledge. A new system for governing distributed knowledge systems is enacted. But such enactment is not enforced by rule of law, as rather by technology and practice that are institutionalized by the license arrangements. This kind of institutionalization forms the background of trust, reliability, and accountability for the interactive system of practical activities (Kogut & Metiu, 2001).

5. Some concluding remarks

In this paper we illustrated how in open-source software projects processes of knowledge making are supported by an underlying web of artifacts of various kinds that play a critical cognitive and organizing role. Although in our study we only examine specific types of artifacts, in large open-source software projects there are thousands of them, all playing with the delicate balance between variety and stability, innovation and conservation of knowledge. They are connected in multiple webs of activities, agents and resources, practices and communications, and are both the outcome and the media of knowledge making. The source code and its multiple beta versions anchor and

circulate software-related knowledge while at the same time inscribing relevant components of human agency and social interaction that facilitate the coordination of a high number of human agents. The code is a generator of variety while being in the focus of activity and attention of a swarm of programmers that keep buzzing around it and are coordinated by its multiple, evolving state(s). Discussion threads on mailing lists support the circulation of project-related knowledge by recording both the log of the development process and the social history of the project. The mailing lists allow for potentially unrestricted access to discussion and at the same time reveal the highly selective structure of the communication threads among multiple programmers discussing a common development issue. The licensing agreements shape an institutional environment that invites experimentation, innovation, and collective problem solving by creating incentives to communication and knowledge sharing. Thus innovation and the search for new knowledge can happen in a dynamic but safe environment where it makes sense to do cooperative work.

Our findings, even at this preliminary stage of research, point to a number of project features that made us conceive of an 'ecology of knowledge'. In such an 'ecology', groups of agents and software artifacts populating the programming environment do not make up for a homogeneous and functionally coherent system, but constitute patched-up and heterogeneous assemblages of components that are constantly recombined, re-shuffled and put to different uses – always being in a dynamic, 'becoming' state. Most of the programmers' innovations originate from creative re-invention and patching up of pre-existing pieces of code and software. Open-source projects enjoy properties of responsiveness, resilience, flexibility and robustness because they can exploit the variety and the redundancy of knowledge resources available in the environment and widely circulated across the Internet.

An open-source software project, when it reaches a critical mass, functions itself as a giant decentralized mechanism for generating and distributing knowledge. The creation of knowledge happens by effectively harnessing distributed knowledge resources for the purpose of the task and, in turn, by distributing new knowledge freely and swiftly through the programming environment.

At the same time dissemination is quasi-automatic, because documentation of built software products or solutions can circulate throughout the web almost instantaneously. Such process is endogenous to the development activity, and is embedded in the Internet-based information and communication infrastructure. Therefore there is little need for organizationally-embedded rules or governance mechanisms. Indeed one of the main points of our study is that governance and coordination mechanisms are largely inscribed in the technology, both in the information infrastructure (the Internet) and in the software tools.

The always changing shape of the software and the code teaches us that knowledge in an open-source software environment is never a final, bounded product, but always in the making, being itself embodied and carried by transient constructs that never reach a completely stable configuration, or perhaps that reach temporary stability but could always be modified in the next round. The transient state of knowledge is nothing but a consequence and a reflection of the underlying structure of interaction characterizing open-source software projects.

Though transient and ever evolving, the artifacts are critical for the creation, accumulation, and dissemination of knowledge in open-source software projects. Indeed, these three activities of knowledge-making do not come in a staged sequence in open-source software projects; instead they are simultaneous and closely interwoven. The processes that allow for the creation of knowledge also contribute to its dissemination, and in turn dissemination is closely linked to creation. Creation and dissemination happen simultaneously, because the communication processes mainly pass along the Internet and are therefore quasi automatically documented in software versions, mailing lists and newsgroups, or other artifacts that are widely circulated across the net. The Internet is the basic infrastructure and repository of knowledge that ultimately makes open-source software development possible by supporting accessibility, interaction and fast circulation of knowledge.

Finally, we are not proposing open-source software projects as a general model of knowledge creation and dissemination in business environments. We think it is a media-specific and contextual phenomenon. Yet, open-source software projects are an interesting subject from

which a number of lessons can be learned. The discovery of the ecological quality of specific

knowledge processes, such as the ones taking place in open-source software projects, can help us to

enrich our current theories about knowledge creation and management, re-consider the meaning and

the relevance of organizational mechanisms for the governance of knowledge systems, and finally

appreciate the generative role of interaction and bricolage in complex design processes.

References

Akrich, M. and B. Latour (1992), A Summary of a Convenient Vocabulary for the Semiotics of Human and Nonhuman Assemblies, in: Bijker, W. and J. Law (eds., 1992), Shaping Technolgy, Building Society: Studies in Sociotechnical Change, Cambridge, Mass: MIT Press, pp. 259-264

Aldrich, H. (1999), Organizations Evoloving, Thousands Oaks, CA: Sage

Anderson, M. and Laird, C. (1988), Evolution and Development of Knowledge Environments, Paper presented at the International Conference on Culture, Language and Artificial Intelligence, Stockholm, May 31 – June 3, 1988

Arthur, M.B., DeFillippi, R.J. and Jones, C. (2001), Project-based Learning as the Interplay of Career and Company Non-financial Capital, in: Management Learning, 32 (1), pp. 99-118

Barley, S.R. and P.S. Tolbert (1997), Institutionalization and structuration: Studying the links between action and institution, in: Organization Studies 18 (1), pp. 93-117

Bateson, G. (1972), Steps to an Ecology of Mind, New York: Ballantine

Baum, J.A.C. and Singh, J.V. (eds.; 1994), The Evolutionary Dynamics of Organizations, New York: Oxford University Press

Blackler, F. (1995), Knowledge, knowledge work and organizations: an overview and interpretation, in: Organization Studies, 16, 6, pp. 1021-1046

Brown, J.S. and P. Duguid (1991), Organizational learning and communities of practice: Toward a unified view of working, learning, and innovation, in: Organization Science, 2, pp. 40-57

Ciborra, C.U. (2002), The Labyrinths of Information: Challenging the Wisdom of Systems, Oxford University Press, Oxford

Ciborra, C.U. and Lanzara, G.F. (1990), Designing dynamic artifacts: computer systems as formative contexts, in: Gagliardi, P. (ed.): Symbols and Artifacts: Views of the Corporate Landscape, Walter de Gruyter, Berlin, pp. 147-165

Ciborra, C.U. and O. Hanseth (1998), From tool to Gestell, in: Information, Technology, and People, 11 (4), pp. 305-327

DeFillippi, R.J, and Arthur, M.B. (1998), Paradox in project-based enterprise: The case of film-making, in: California Management Review 40 (2), Winter 1998

DeFillippi, R.J. (2001), Project-based Learning, Reflective Practices and Learning Outcomes, in: Management Learning, 32 (1), pp. 5-10

DeSanctis, G. and P. Monge (1998), Communication Processes for Virtual Organizations, in: Journal of Computer Mediated Communication 3 (4), June 1998

Dixon, N. (2000), Common Knowledge, Harvard Business School Press, Cambridge, MA.

Eisenhardt, K.M. and Santos, F.M. (2002), Knowledge-Based View: A New Theory of Strategy?, in: Pettegrew, A., Thomas, H., and Whittington, R. (eds.), Handbook of Strategy and Management, London: Sage, pp. 139-164

Engeström, Y. (1997), Learning by Expanding: An Activity Theoretical Approach to Developmental Work Research, Helsinki: Orienta Konsultit

Eveland, J.D. and T.K. Bikson (1988), Work Group Structures and Computer Support: A Field Experiment, in: ACM Transactions on Information Systems, Vol. 6, No. 4, October 1988, pp. 354-379

Goffman, E. (1983), The Interaction Order, in: American Sociological Review 48 (1983), pp. 1-17

Goodman, P.S. and L.S. Sproull (1990), Introduction, in: Goodman, P.S. and L.S. Sproull and Assouciates (eds., 1990), Technology and Organizations, San Francisco, CA: Jossey-Bass

Grandori, A., B. Kogut, and A. Lewin (eds., 2002), Special Issue on 'Knowledge, Knowing, and Organizations', Organization Science, Vol. 13, No. 3

Grant, R.M. and J.-C. Spender (eds., 1996), Special Issue on 'Knowledge and the Firm', Strategic Management Journal, Vol. 17, Winter 1996

Hanseth, O. (1996), Information Technology as Infrastructure, Ph.D. Dissertation, Department of Informatics, Goteborg Universitet, Report 10, November 1996

Harhoff, D., Henkel, J. and von Hippel, E. (2000), Profiting from Voluntary Information Spillovers: How Users benefit by freely revealing their Innovations, MIT Sloan School of Management WP 4125, July 25, 2000

Healy, K. and A. Schussman (2003), The Ecology of Open-Source Software Development, Working Paper, Department of Sociology, University of Arizona, Social Sciences, 2003

Iannacci, F. (2002), The Social Epistemology of Open-Source Networks, Working Paper, Department of Information Systems, London School of Economics and Political Science, 2002, pp.1-28

Joerges, B. and B. Czarniawska (1998), The Question of Technology, or How Organizations Inscribe the World, in: Organization Studies, 1998, 19(3), pp. 363-385

Kling, R. and W. Scacchi (1982), The Web of Computing: Computer Technology as Social Organization, in: Advances in Computers, 21, pp. 1-90

Kogut, B. and A. Metiu (2001), Open Source Software Development and Distributed Innovation, in: Oxford Review of Economic Policy, 17 (2), pp. 248-264

Kogut, B. and U. Zander (1992), Knowledge of the Firm, Combinative Capabilities, and the Replication of Technology, in: Organization Science, Vol. 3, No. 3, 1992, pp. 383-397

Kollock, P. (1999), The Economies of Online Cooperation: Gifts and Public Goods in Cyberspace, in: M. Smith and P. Kollock (eds., 1999), Communities in Cyberspace, London: Routledge

Kuwabara, K. (2000), Linux: A bazaar at the edge of chaos, in: First Monday, 5 (3), accessible at: http://firstmonday.org/issues/issue5_3/kuwabara/

Lakhani, K. and Hippel, E. von (2000), How Open Source software works: "Free" user-to-user assistance, MIT Sloan School of Management Working Paper #4117, May 2000

Lanzara, G.F. (1997), Self Destructive Processes in Institution Building and some Modest Countervailing Mechanisms, in: European Journal of Political Research, 33, pp. 1-39

Lanzara, G.F. (1999), Between transient constructs and persistent structures: designing systems in action, in: Journal of Strategic Information Systems 8 (1999), pp. 331-349

Latour, B. (1992), Technology is society made durable, in: Law, J. (ed., 1992), Sociology of monsters: Essays on power, technology and domination, London: Routledge, pp. 103-131

Latour, B. and S. Wolgar (1979), Laboratory life: The construction of scientific facts, Princeton: Princeton University Press

Lee, G.K. and R.E. Cole (2000), The Linux Kernel Development as a Model of Knowledge Creation, Working Paper, Haas School of Business, University of California, Berkeley

Lerner, J. und Tirole, J. (2001), The Open Source Movement: Key Research Questions, European Economic Review, Vol. 46 (2001), S. 819-826

Luhmann, N. (1975), Interaktion, Organisation, Gesellschaft, in: Luhmann, N. (ed., 1975), Soziologische Aufklärung II, Opladen, S. 9-20

Luhmann, N. (1995): Social Systems, Stanford: Stanford University Press

Luhmann, N. (1997), Die Gesellschaft der Gesellschaft, Frankfurt am Main 1997

Luhmann, Niklas (2000): Organisation und Entscheidung, Opladen/Wiesbaden: Westdeutscher Verlag

March, J.G. (1991), Exploration and exploitation in organizational learning, in: Organization Science, 2, pp. 71-87

McGowan, David (2001): Legal Implications of Open Source Software, in: University of Illinois Law Review, Vol. 1 (2001), pp. 241-304

Metiu, A. and B. Kogut (2001), Distributed Knowledge and the Global Organization of Software Development, Working Paper, accessible at: www.opensource.org

Miles, M.B. and A.M. Huberman (1984), Qualitative Data Analysis: A Sourcebook of New Methods, Newbury Parc, CA: Sage

Mockus, A., R.T. Fielding, and J. D. Herbsleb (2002), Two Case Studies of Open Source Software Development: Apache and Mozilla, in: ACM Transactions on Software Engineering and Methodology (TOSEM), July 2002, Vol. 11, Issue 3; accessible at: http://www.research.avayalabs.com/techreport/ALR-2002-003-paper.pdf

Moglen, Eben (1999): Anarchism Triumphant: Free Software and the Death of Copyright, in: First Monday, Peer-reviewed Journal on the Internet, Vol. 4, October 1999, http://firstmonday.org/issues.html (edited 2nd August 2001)

Morner, M. (2003), The Emergence of Open-Source Software Projects: How to Stabilize Self-Organizing Processes in Emergent Systems, forthcoming in: Hernes, T. und Bakken, T. (eds.; 2003), Autopoietic Organization Theory: Drawing on Niklas Luhmann's Social System Perspective, Abstrakt Forlag, Oslo

Neff, G. and D. Stark (2002), Permanently Beta: Responsive Organization in the Internet Era, forthcoming in: P.E.N. Howard and S. Jones (eds., 2002), The Internet and American Life, Thousands Oaks, CA: Sage, 2003

Nonaka, I. and H. Takeuchi (1995), The Knowledge Creating Company, New York: Oxford University Press

Orlikowski, W.J. (1992), The Duality of Technology: Rethinking the Concept of Technology in Organizations, in: Organization Science, 3, 3, S. 398-427

Patriotta, G. (2003), Sensemaking on the shopfloor: Narratives of knowledge in organizations, in: Journal of Management Studies, Vol. 40, No. 2, pp. 349-375

Patriotta, G. and G.F. Lanzara (2001), The making of a factory: Dynamics of institution building at Fiat's Melfi plant, Paper presented at the Annual Meeting of the Academy of Management, Washington 3-7 August

Patriotta, G. and G.F. Lanzara (2003), The Inscription of Agency into Institutions, Paper presented at EGOS Workshop "Institutional Change", EGOS Colloquium, Copenhagen, July 2003

Raymond, E.S. (2001), The Cathedral and the Bazaar: Musings on Linux and Open Source from an Accidental Revolutionary, Sebastapol, CA: O'Reilly and Associates

Scacchi, W. (2001), Understanding the Requirements for Developing Open Source Software Systems, accepted for publication with revisions in: IEE Proceedings – Software, Paper No. 29840, December 2001

Schelling, T.C. (1960), The Strategy of Conflict, Harvard University Press

Simon, H.A. (1969), The Sciences of the Artificial, Cambridge: MIT Press

Sindig-Larsen, H. (1987), Artificial Intelligence and the Ecology of Knowledge. Some Background Ideas for the Programme Committee of the Conference "Culture, Language and Artificial Intelligence", Stockholm, May 31 – June 4, 1988

Spender, J.-C. (1996), Competitive advantage from tacit knowledge?: Unpacking the concept and its strategic implications, in: Moingeon, B. and A. Edmondson (eds., 1996), Organisational Learning and Competitive Advantage, London: Sage

Spiller, D. and T. Wichmann (2002), Basics of Open Source Software Markets and Business Models, FLOSS Final Report – Part 3, Berlin: Berlecon Research

Stallman, Richard (1998): The GNU-Project, accessible at http://www.gnu.org/gnu/the-gnu-project.html, 1998 (edited 27th November 2001)

Strauss, A. and J. Corbin (1990), Basics of qualitative research, Thousands Oaks, CA: Sage

Swan, J. and H. Scarbrough (eds.; 2001); Special Issue on Knowledge Management, Journal of Management Studies, Vol. 38, Issue 7

Swan, J. and H. Scarbrough (2001), Knowledge Management: Concepts and Controversies, in: Journal of Management Studies, Special Issue on Knowledge Management, 38, 7, pp. 913-921

Tuomi, I. (2000), Internet, Innovation, and Open Source: Actors in the Network, Working Paper, SITRA, The Finnish National Fund of Research and Development, 2000

Von Hayek, F.A. (1945), The Use of Knowledge in Society, in: American Economic Review, Vol. 35, No. 4, September 1945, pp. 519-530

Von Hippel, E. (2001), Innovation by User Communities: Learning from Open-source Software, in: MIT Sloan Management Review, Vol. 42 (2001), pp. 82-86

Von Hippel, E. and von Krogh, G. (2003), Open source software and the private-collective innovation model: Issues for organization science, in: Organization Science, Vol. 14, No. 2, 2003, pp. 209-

Von Krogh, G., S. Spaeth, and K.R. Lakhani (2003), Community, Joining, and Specialization in Open Source Software Innovation: A Case Study, forthcoming in: Research Policy Special Issue on Open Source Software Development (2003)

Von Krogh, G. (2002), The communal resource and information systems, in: Journal of Strategic Information Systems 11 (2002), pp. 85-107

Weick, K. E. (1995), Sensemaking in organizations, Thousands Oaks, CA: Sage

Weick, K.E. (1990), Technology as equivoque: Sensemaking in new technologies, in: P.S. Goodman and L.S. Sproull (eds., 1990), Technology and Organizations, pp. 1-14

Zipf, G.K (1949), Human Behavior and the Principle of Least Effort, Reading, MA: Addison-Wesley Publ. Co.